Protecting User Privacy from Web Tracking Threats

Øyvind Sikkeland

Thesis submitted for the degree of
Master in Informatics: Programming and System Architecture
(Information Security)
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2020

Protecting User Privacy from Web Tracking Threats

Øyvind Sikkeland

Protecting User Privacy from Web Tracking Threats

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Web trackers are everywhere on the Internet today, and allows websites to collect information about users in order to uniquely identify them, and link recurring visits. Information about users' browsing behaviour, geographical location, details about their computer, and much more, can be collected and potentially misused. This research aims to protect user privacy from these web tracking threats.

This thesis describes the design and implementation of a protecting framework. This system is implemented as a web browser extension, with two main countermeasures. The first countermeasure relies on blacklists, which contains signatures of known web trackers. When HTTP requests are made by the browser, this approach detects the requests and matches the URL in the request with signatures in the blacklist. If a match is found, the request is cancelled and no further communication can be made between the browser and the website. More importantly, the website is not able to track the user. The second countermeasure is directed at cookies. This approach detects domains that use cookies in a third-party context, and based on the prevalence, potentially takes action and removes cookie related headers from HTTP messages to these domains.

Testing showed that this tool blocked more web trackers than several state of the art tools, but it was beaten by the Brave browser. A limitation in this evaluation, was that the tools were tested on only 30 websites. A bigger sample size could provide more conclusive results.

In conclusion, the tool accomplished its goal to protect user privacy from many web tracking threats. However, web trackers are continually improved and changed, which makes room for more research, and potentially better countermeasures.

Contents

List of Figures

List of Tables

Preface

Acknowledgements

I would like to express my gratitude to my supervisor Dr. Nils Gruschka for all his invaluable guidance, feedback and support throughout this project. Thank you!

I would also like to thank Guleed Abdi for the great cooperation on the topic of web tracking.

Lastly, I wish to thank my family for all the support, and for always believing in me. A special thanks go to my dear Guri, for her continuous support and encouragement throughout this process.

Chapter 1

Introduction

1.1 Motivation and Problem Statement

The Internet is becoming a bigger and bigger part of everyone's life. We use it for everything from banking and shopping, to entertainment. We give up a lot of our personal information to web services. Often voluntarily, e.g. by filling out a web form when creating an account or buying something. However, much more information about us is gathered "under the hood" and is completely transparent to users. Many people have probably noticed when searching for something to buy, maybe a new pair of shoes, and when they later go to their Facebook page, they are shown advertisements for the exact same product they were previously searching for. This is possible if websites are able to collect uniquely identifying information about users, that lets them link recurring visits - a process called *web tracking* [103].

There are many available techniques to collect this information about users, for instance analysis of IP headers, HTTP messages, and even using JavaScript and Flash; often a combination of several techniques [20]. The information collected can be IP addresses, details about the client's operating system, browser and hardware, and even users' geographical location and browsing history [20].

Web tracking is mainly employed to display personalized advertisements and product recommendations to users, however, research have found that web tracking is used for more sinister purposes, such as price discrimination [69]. In later years, web tracking has raised concerns in both the general public and in the research community [103], and much research has been conducted in the last decade to get an overview of the techniques used to track Internet users, as well as how to protect against it. Several defence tools have been developed to combat this rising issue and relies on different protection mechanisms, however, this is an on-going battle. As new tracking mechanisms are developed and employed, the countermeasures must adapt as well.

The problem is that user privacy on the Internet is generally not respected, and even though GDPR and the ePrivacy Directive [44] requires websites to get users' consent before using some tracking techniques, there

exists other tracking techniques that does not require consent by users and are completely transparent. In order to protect user privacy, programs that block or limit the ability of web tracking techniques are needed.

1.2 Thesis Content and Contribution

The topic of web tracking is divided into two theses, and partly involves a cooperation between Guleed Abdi and Øyvind Sikkeland. The task of this thesis is to analyze the most common methods used in web tracking as well as the state of the art in protecting from web tracking in theory and practice (e.g. Firefox, Disconnect, Ghostery). Further, a unified protecting framework shall be developed, implemented and evaluated.

Abdi will research the state of the art existing web tracking methods, survey the largest websites on the Internet today for what tracking tools they use, and build a custom service to implement these trackers in order to analyse how they work and what data is tracked.

My contribution is an overview of the most common web tracking methods, including how they work and what kind of information they gather about the user. Secondly, it presents different protection methods, and pros and cons with these methods, as well as an analysis of how the state of the art tools work. Further, the development of a protecting framework and a following evaluation is conducted.

1.3 Research Questions

The thesis will elaborate on these research questions:

- What kind of countermeasures to web tracking exist today?
- How does the state of the art protection tools available to Internet users today protect their privacy?
- A new defense tool - how does this compare to the state of the art?

Chapter 2

Background

2.1 What is Web Tracking?

Web tracking techniques were first developed to enhance the user experience when browsing the web. It's used for user authentication and remembering login-information, shopping carts and similar [103], and also to check browser configurations to, for instance, make a website fit a particular screen resolution. However, because this often involves gathering information about users without their knowledge and consent, it has in later years raised concerns about users' privacy on the web both in the research community and in the general public [103]. Using different tracking techniques, which will be explained later in this chapter, a personal profile can be created for each visitor. These profiles can be up to 100% accurate, and be directly tied to an advertising network or sold to advertisement companies.

Web tracking today is almost everywhere. Roesner et al. [101] found in their research that there were at least one tracker monitoring each website on the Alexa top 500 websites (but most of the times several trackers), and they also found more than 500 unique trackers in the wild. Due to the extent of web tracking today, information about users' financial situation, interests and shopping plans can be gathered, but also conclusions about a users' sexual orientation, health, political views and religious beliefs can be drawn [65]. When the user logs into one of their private accounts, for instance their Facebook account, this information can be linked directly to their name. [65]

Some research have also found that web tracking is used for even more sinister purposes. Research from 2014 [69] found that web tracking is used for price discrimination. The researchers found that the price for a hotel room, for instance, can be up to 20% different for different users, based on the information the web service can gather from the client. They also discovered that tracking techniques are used to alter search results on sites like Google, based on the users' browsing history and purchases, to make the results more personalized. Surveillance by the government is also a concern. NSA is collecting a lot of information, and this has been found to also include unique user ID's generated by third-party trackers as well as

geolocation information [72].

2.2 First-Party and Third-Party Tracking

There is a clear distinction between first-party and third-party tracking. First off, a first-party website is a website directly visited by a user, while a third-party website is an external site connected to the first-party in some way, for instance by having elements embedded in the first-party website as advertisement banners [68]. Tracking techniques used by a first-party website are much more often used to enhance user experience. Cookies can be used to save shopping carts or set the appropriate language, and fingerprinting techniques can be used to change the appearance of the web screen to fit the user's screen resolution. In contrast, third-party trackers are considered a huge threat to user privacy on the web. They are generally connected to a vast amount of other websites and transparently gathers information about a user across the web [20].

2.3 Web Tracking Techniques

Web tracking fundamentally relies on mechanisms that can identify visitors and re-identify them on subsequent visits. This started out in 1994 when Lou Montulli at Netscape [90] developed cookies. Clients and servers communicate over the *HTTP protocol* which by itself is stateless. Therefore they needed a mechanism to recognize when a user came back to visit a website. The purpose behind this was to enhance the user experience by letting the website know where a user left off when they left the website, and then resume interaction at that point when they came back [110]. *HTTP cookies* were thereby created, to allow the HTTP protocol to preserve state information [110]. They were originally not intended for web tracking purposes, but today they are the most common tracking method [43].

Cookies are limited in how much data they can store, and have evolved immensely since 1994. Flash cookies were created and is set by the Flash plugin. They can store much more data, circumvent browser privacy settings and can be used to respawn deleted HTTP cookies [113]. In later years, cookies have continued to evolve, and new storage mechanisms, some introduced by HTML5, can be used to store state on a user's device. The evercookie [82] can be utilized to store the information in multiple storages, and respawn those that are deleted.

In recent years we have also seen a rise in *stateless* tracking techniques being used to track Internet users. Most of these techniques are also called *fingerprinting* techniques, and is a collection of several methods to collect information about users' machine or browsing behaviour that does not rely on storing any data on the user's device. Instead, the information is collected when a user visits a website and stored as a profile by the web server. When the user re-visits, the information is again collected. If everything matches a previously stored profile, the user has been re-identified. If enough methods are combined, they can together potentially

make up a unique fingerprint [94]. Fingerprinting mainly relies on client-side scripts, like JavaScript, and available APIs to gather this information, but also information from HTTP messages are used to make up the fingerprint. These methods were proposed by researchers as potential privacy risks many years ago [95], and today they are widespread on the Internet as an alternative to cookies.

2.3.1 Stateful Tracking Techniques

Stateful techniques needs to store state, basically some kind of data, on the user's device. This can be achieved in several ways, but the most common method is *cookies*.

In this section we will look at common techniques to store state on users' devices, and methods to share this information back to the tracker, or to other trackers.

Client-Side Storages

Stateful tracking techniques relies on utilizing client-side storages to store data. Client-side state can include different things like audio and video, but browser cookies are the most common form of tracking method. HTTP cookies are stored in a file by the browser, and can be found in the browser folder on the computer. But there are more storages that websites can use to store state on clients' machines. HTML5 introduced completely new client-side storage mechanisms for browsers. Using the *Web Storage API* [124] grants websites the ability to store data, like cookies. Most notably the *sessionStorage* and *localStorage*. A JavaScript script running in the context of a first-party or third-party website can easily store cookies in these storages [20]. The session storage lets the website store data for the duration of the page session, meaning as long as the browser is open [124]. This storage limit is at most 5 MB, which can contain much more information than a normal HTTP cookie. However, the local storage is the most used because data stored there persists even when the browser is closed. The stored data has no expiration data and can only be cleared with JavaScript or clearing the browser cache or locally stored data [124].

The *IndexedDB API* [79] is another API that can be utilized to store cookies. This provides the browser with a complete database system and is generally used to store complex data. This storage mechanism isn't widely used for cookies, but has been found to play a role in evercookies, which is explained in the section *Evercookies*.

Moreover, the Flash plugin introduces another storage mechanism which lets websites that embed content set *Local Shared Objects*, also called Flash cookies, on the client's machine [113].

HTTP Cookies

HTTP cookies are small text files containing some information. They can store up to 4 KB of data. These cookies are usually handled and stored by

the web browser. A web server can send cookies to a client using the HTTP protocol with the *Set-Cookie* response header [104]. The client's browser will send the cookie back to the web server with the *Cookie* header [42]. Cookies can also be set or read with JavaScript using the *Document web API* [54]. Using the *document.cookie* property a website can create, read and delete cookies. This means that a client-side script and HTTP can work together to handle cookies for a website. However, if the *HttpOnly* flag of a cookie is set, client-side scripts cannot access the cookies set via HTTP [74].

A cookie must have a name and a value, but they can also contain more attributes with information about the cookies expiration, domain and flags. Cookies have many uses, but according to Mozilla [74] they have three main purposes:

- ***"Session management***
  - *Logins, shopping carts, game scores, or anything else the server should remember*
- ***Personalization***
  - *User preferences, themes, and other settings*
- ***Tracking***
  - *Recording and analyzing user behavior"* [74]

This research focuses on the tracking purpose of cookies.

When cookies are created, the creator can specify an expiration time as mentioned above. If nothing is specified, the cookie is created as a *session cookie* and will expire and be deleted when the client's browser is closed. However, browsers can use *session restoring*, which makes the session cookies permanent. If an expiration date is specified, the cookie is also treated as a permanent cookie [74].

An important part to HTTP cookies is that they are domain-specific [74]. This means that a cookie set by *example.com* cannot be sent to another domain, e.g. *tracker.com*. Cookies have a domain attribute that can be specified, and if left unspecified, the cookie can only be sent to that specific domain, excluding subdomains. If the domain attribute is set to *example.com*, the cookie can also be sent to subdomains, like *example2.example.com*. This helps with security and, in theory, privacy as well. The same-origin policy [102] prevents cookies (and also other documents and scripts) from being loaded from a third-party context into a first-party context, which means a third-party tracker can not read cookies set by another first-party domain. But trackers have found a way to circumvent this policy by "leaking" information, for instance in URLs. The mechanism called *cookie synchronization* [98] will be explained more in detail later in this section.

HTTP cookies used for tracking usually have names like *uid* and *userid* which contains a value that is unique for every user. There are several variations of these names, but an example of the prevalence of specifically the uid cookie can be found here [121]. This search finds that uid has been found as a third-party cookie on 128,418 websites with an average life span

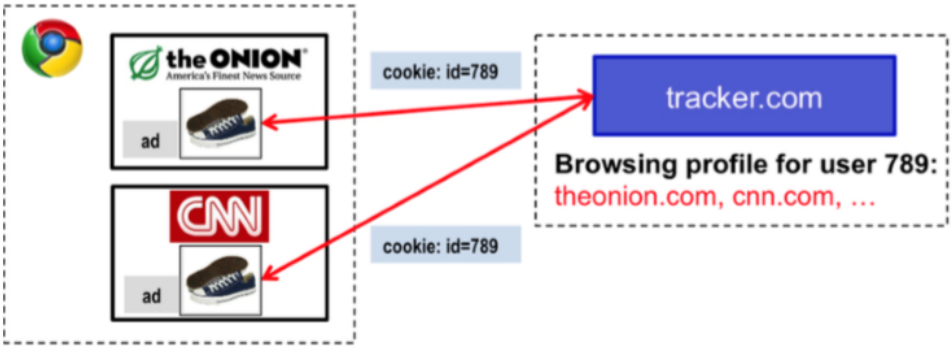


Figure 2.1: Third-party cookie-based tracking. Both the Onion and CNN shows ads which connects to the same tracking website (tracker.com). Figure taken from [14].

of 213,393 days (as of 02/04/20). This indicates that HTTP cookies used for tracking purposes is an extremely widespread method.

An example to illustrate how a third-party tracking service can gather data about a user's browsing behaviour and history across several websites can be seen in Figure 2.1. Because of the same-origin policy, if a user visits *theonion.com* which sets a cookie, a third-party domain *tracker.com* can not see this cookie by default. The way around this is to let the third-party domain (*tracker.com*) set the cookie. So we can see that if a user visits *theonion.com*, which embeds an ad from the *tracker.com* domain, *tracker.com* can set a unique cookie (in this example with an id=789) for the user. When the user later visits *cnn.com*, which also embeds an ad from *tracker.com*, the same cookie is sent back to *tracker.com*. The tracker now knows that the user has visited both websites [14]. The tracker can store a browsing profile for this user, and store their browsing history. They may also be able to log what the user clicked on, how many times they visit each website and how long they stayed on the sites. If a tracker is connected to enough websites, they can potentially know everything about a user's browsing habits without they knowing anything about this. If the tracker is only connected to "anonymous" websites, meaning where the user isn't logged in or shared their real name, the tracker only knows the user by the unique identifier. However, if the tracker is also connected to e.g. Facebook, they will know the user's personal name as well as their browsing history [65].

Cookie Synchronization

Cookie synchronization, often shortened to cookie syncing, isn't a tracking mechanism by itself, but rather a method to circumvent the same-origin policy and help trackers share stateful information, like cookies [98]. Trackers that are connected to different tracking networks will create different unique identifiers for users. This is a problem for advertising companies if their ads aren't embedded by many websites. Using the cookie synchronization technique, it's possible for different trackers to share their unique

user identifiers and synchronize these, in order to have the same unique user identifiers, that allows both companies to track users across even more websites [98]. To illustrate how this technique could work in practice, let's look at an example below. The idea for this example and illustration is provided in [98].

Step 1 is illustrated in Figure 2.2. The user visits *site1.com* that embeds a tracking script from *tracker1.com*. To load this script, a request is made by the browser to *tracker1.com*. A response is sent to the user's browser containing a *Set-Cookie* header with a cookie value *123*.

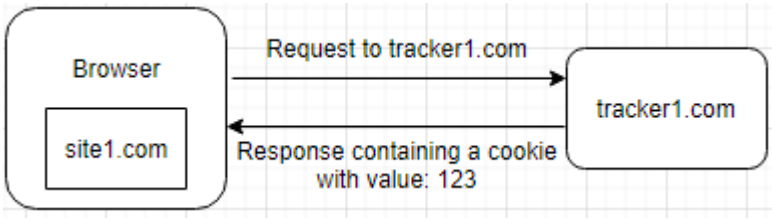


Figure 2.2: Example of how cookie synchronization can work.

Step 2 is illustrated in Figure 2.3. The user visits another website, *site2.com*, that embeds a tracking script from another tracker, *tracker2.com*. The user gets another cookie value for this website, *456*.

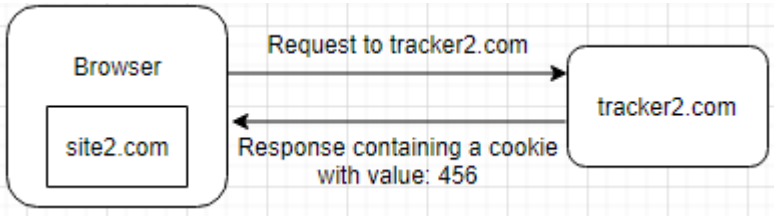


Figure 2.3: Example of how cookie synchronization can work.

Step 3 is illustrated in Figure 2.4. This is where the synchronization between *tracker1.com* and *tracker2.com* happens. The user visits another website, *site3.com*, that embeds a tracking script from *tracker1.com*, but not *tracker2.com*. Therefore, *tracker2.com* wouldn't know that the user visits *site3.com*. However, when the user's browser makes a request to *tracker1.com*, this site responds with redirect request, which forces the user's browser to make a request to *tracker2.com*. This request can be constructed with a custom URL containing several parameters, that can include both the user's unique cookie for *tracker1.com* and information about *site3.com*. *tracker2.com* now knows that the user they know by cookie value *456* has visited *site3.com*, and they learn that the user *tracker1.com* knows by cookie value *123* is the same user as they know by the cookie value *456* [98].

After step 3 is completed, the two trackers can perform back-end data merges [98].

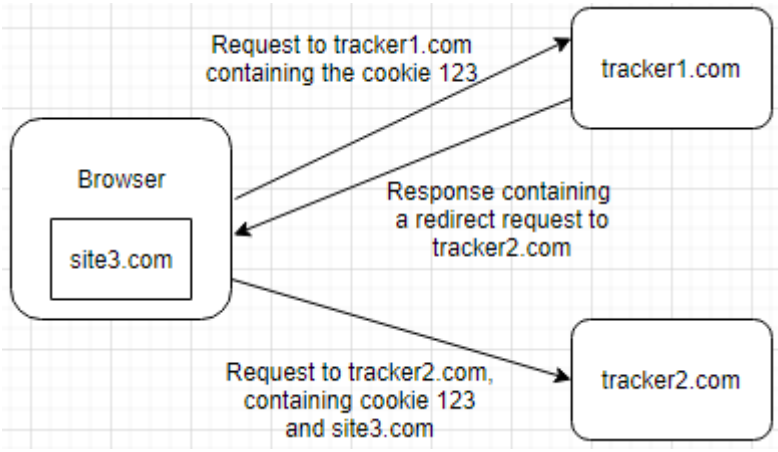


Figure 2.4: Example of how cookie synchronization can work.

Cookie synchronization was first observed in use in 2012 [20], and research has found that its usage is extremely widespread today. Roesner et al. [101] observed a *"large number of cookie leaks"* [101] in their analysis in 2012. Later research done by Papadopoulos et al. [98] in 2019 found that 157 out of the top 200 websites are connected to third-parties which are involved in cookie synchronization with at least one other party. They are also capable of reconstructing 62-73% of a user's browsing history.

Supercookies

Supercookies originally referred to cookies with only a top-level domain (e.g. *.com*) and were not domain-specific (e.g. *google.com*) [7]. This was a huge privacy and security concern because it meant that cookies not set by *example.com* could be sent to it from any *.com* domain. Most browsers now block supercookies by default by using the *Public Suffix List* [100], which makes these types of cookies not much of a concern anymore.

Today, supercookies generally refer to something else. They can be described as techniques or procedures that encompass three different mechanism [24, 41]:

- **Collecting information about users in various ways.** This can include both stateful and stateless methods.
- **Utilizing different storage mechanisms.** Depending on the client, this can include browser storages, Flash cookie storages, and many of the storages introduced by HTML5.
- **Helping to respawn other cookies.** Supercookies are mainly used to respawn *zombie cookies*. This generally refer to deleted HTTP cookies that are respawned after deletion.

The most widespread supercookies today are *Flash cookies* and *evercookies*. They will be elaborated on below.

Flash Cookies/Local Shared Objects

Flash cookies were first identified in 2009 [20], and were the strongest stateful tracking technique after the basic HTTP cookies. They are similar to HTTP cookies in that they can be used by websites to collect information about users. Flash cookies were originally created with users in mind - just like with HTTP cookies. They are utilized to enhance user experience by storing user preferences, volume of Flash videos, save data from Flash games among other things [113]. They are stored on clients' computers as *local shared objects*, which is the technical term of the Flash cookie. This is a text file that is created by the Adobe Flash plugin. When a website embeds a Flash element, for instance an advertisement banner, a request is made by the client's browser for this resource, and the text file (cookie) is sent to the client. Websites can embed Flash elements from third-parties as well, allowing the Flash cookie to track users cross-site. When they are used for tracking purposes, they contain a name and a value, where the value is unique for each user. Flash cookies used in web tracking were found to often have the name *uid* or *userid*, and contain a value of 16 or 32 bits [113].

What makes the Flash cookie particularly threatening, is the supercookie mechanisms. They are far more persistent than HTTP cookies and can contain up to 100 KB of information, compared to HTTP cookies' 4 KB limit [113], and they have no expiration date by default. Flash cookies are stored in users' local file system, the Flash storage, where browsers have no control. If users have enabled browser privacy settings, such as automatically clearing browser cache and history on browser exit, it has no effect on the Flash cookie. The fact that Flash cookies are stored on the client's local file system, makes it possible for the cookie to track user's across different web browsers as well.

Another use of Flash cookies is to act as a backup for HTTP cookies. If a website set both an HTTP cookie and a Flash cookie with the same name and value on a user's machine, and the user clears their browser cookie storage, the Flash cookie can respawn the HTTP cookie [113]. This is very problematic and a significant privacy concern for users who do do not wish to be tracked and deletes HTTP cookies stored by their browser, as they are still being tracked by the Flash cookie which respawns the HTTP cookie [113]. Flash cookies were also found to be able to track users in *private browsing mode* [113]. All of these mechanisms combined makes the Flash cookie a type of supercookie, and more effective in web tracking than HTTP cookies.

Soltani et al. [113] surveyed the websites on the Alexa top 100 list in 2009, and Ayenson et al. [10] performed a follow-up study in 2011. In 2009 they found that 54 out of 100 websites used Flash cookies, and a total of 281 Flash cookies were found. In 2011, the numbers had dropped significantly. Only 37 websites used Flash cookies at all, with a total number of 100 cookies. This indicates that Flash cookies are decreasing in use. A reason for this is that Flash usage by websites have dropped from being used on 28.5% of websites in 2011, to only 4.9% in 2018 [38]. Both research papers [10, 113] also studied the prevalence of respawning HTTP cookies

using Flash cookies. In 2009, they found that 6 out of the top 100 websites utilized this technique, while the numbers had dropped to only 2 out of the top 100 websites in 2011. This indicates that respawning HTTP cookies using Flash cookies isn't a widespread technique.

Evercookies

Evercookies [82] were created and presented by Samy Kamkar in 2010, and often referred to by researchers as a type of supercookie. Evercookie is a JavaScript API to produce extremely persistent cookies. Evercookie works by storing cookie information in many different types of storage mechanisms, which includes HTTP cookies and ETags[1] [56], web history and cache, Flash cookie storage, HTML5 storages, and even storing the cookies as RGB values which can be read back out using HTML5 Canvas tag [82]. If the user clears some of these storages, the evercookie will recreate the deleted cookies in these storages. This means that the evercookie only needs to be present in one storage to recreate the cookie in all storages. If Flash Local Shared Object, the Silverlight Isolated Storage or specific Java mechanisms are available, the evercookie can reproduce in other browsers on the same client machine [82]. This makes the evercookie exceedingly difficult to remove by users. However, in some clients, some of these storage mechanisms might not be supported, which renders the evercookie less effective.

The survey performed in 2011 by Ayenson et al. [10] was the first study which found the use of browser cache and HTML5 storage mechanisms as evercookies by websites in the wild. However, they discovered that only two websites out of the Alexa top 100 were respawning any cookies (one of them used both Flash cookies, browser cache and HTML5 storage mechanisms for respawning).

In 2014 another study were conducted by Acar et al. [3] to detect the use of evercookies and respawning. In this study, they only tried to detect the use of Flash evercookie; meaning, in the cases where the Flash cookie storage mechanism were used to respawn other cookies. They analysed the top 200 popular websites and found that 10 of these utilized Flash evercookies to respawn other cookies. They also ran the automated analysis on the top 10,000 websites and found 33 different Flash cookies, which were used to respawn more than 175 HTTP cookies on 107 of the websites. Even though many storage mechanisms are used by the evercookie, it's likely that most utilize Flash cookies because they have a big advantage compared to the others: Flash cookies can be shared between different browsers who make use of the Adobe Flash plugin.

In the same study conducted by Acar et al. [3], the researchers also did another interesting discovery. One of the storage mechanisms evercookie can utilize is the *HTML5 IndexedDB* [78], but this was the first report of IndexedDB being used as an evercookie vector. The number of websites that used this was very small, however, with only 20 out of the top 100,000

[1]The ETag is a HTTP response header with caching capabilities and can be misused in web tracking.

websites.

Because of the privacy concerns and difficulty in removing evercookies, this has gotten some attention in the media. In 2011 the analytics company *KISSmetrics* was sued because of their alleged use of supercookies and respawning techniques to track people, and finalized the settlement of a class-action lawsuit in 2013 [45].

Cache-Based Tracking Techniques

Previously in this chapter, we looked at various storages that are used to store state, such as cookies. Caches are smaller types of storages, mainly used to store temporary data that are accessed regularly to shorten the processing time. However, caches can also be used to collect information about users.

The most common method is misusing browser's web cache to gain knowledge about what websites users visit [20]. Here is an example of how this works: the web cache stores certain web elements in order to avoid having to download them again to shorten the loading times. If a website embeds an advertisement in the form of an image, this image is stored in users' web cache. On subsequent visits, trackers can check whether this image is downloaded again or not. If it's not, the tracker knows that the user's browser has cached it before. If the same advertisement is present on many websites, the tracker can detect which websites the user has visited [20]. There are mainly three techniques used to exploit the web cache:

- *Embedding identifiers in cached documents:* Websites can force the user to request a particular HTML page. Within this HTML page, a unique identifier can be stored, for example in an HTML *div* property. When this page is cached, the identifier can be read by the website. If a tracking company have this HTML page embedded in large numbers of websites, the result is that all these websites can access the unique identifier from the user's web cache [20].

- *Loading performance test:* JavaScript can be used to measure the load time of different objects. If a website has an image, this is loaded and placed in the cache by the browser. On subsequent visits, a script can measure the time it takes for this image to load. If it's cached, the load time will be significantly shorter than if it's not. In this way, a tracker can detect whether a user has visited the website before [20].

- *Entity Tags (ETags) and Last-Modified HTTP headers:* When an object is first loaded by the browser, the HTTP header contains these fields: *Last-Modified*, *ETag*, *Cache-Control* and *Expires*. It has been found that the *Last-Modified* and *ETag* fields can be used to store unique identifiers which are then cached by the browser. These fields accept random strings, and is of sufficient size to store a unique identifier. When a user's browser has cached an object like this, and then visits the website later, the browser sends two different HTTP headers that contains the values from the *Last-Modified* and *ETag* header fields.

The web server checks whether these values are outdated or not, and if they are not, the server responds with an *HTTP 304 Not Modified* status. If they are outdated, the website responds with a new document [20]. Using this method, a website can store unique identifiers in browser's web cache, and track users.

2.3.2 Stateless Tracking Techniques

Stateless tracking techniques refer to techniques that does not need to store data on users' device. Most of the techniques that falls under this category, are also called fingerprinting techniques. These can be seen as a group or collection of several different methods. They are less intrusive and harder to defend against than stateful techniques, because they do not need to store any information on the device [14]. These methods gather different kinds of data, such as user and device specific settings and configurations in order to make a unique fingerprint for that device. This means that instead of, for instance, checking a cookie, the tracker will have to check each method used for making the fingerprint each time the user visits the website in order to re-identify the user. This process is completely transparent to the user, and web services that use these techniques are not required to notify the user [103]. This means the user has not given their consent, and therefore a serious privacy threat. Even if the user could know, there is no simple way of op-ting out [94].

Client-side scripts like JavaScript have access to different APIs with read access to several operating system and browser settings/configurations, which means tools are easily available for generating fingerprints. Java and Flash is also utilized for making fingerprints. This means that if a user disables Java or Flash, but in particular JavaScript, in their browser, they can somewhat limit these tracking techniques, but will render most of the web unusable. However, it still isn't enough to prevent passive fingerprinting [20]. Passive fingerprinting doesn't rely on any code to be executed, instead it's about observing the details in web requests, and make the fingerprint based on those characteristics.

An open-source library called *fingerprintjs2* [57] is a popular script that websites can use to fingerprint its users. It can collect a lot of information about a user, which are used to create a hash that acts as the fingerprint. The collected information is illustrated in Figure 2.5 when used on my own computer (the fingerprint test can be conducted on *www.ddzero.net)*.

This section will further elaborate on the available stateless techniques.

HTTP Referer Header

The HTTP *referer* request header [75] can be used as part of a fingerprint. Because it doesn't need any code to be executed, it falls under the passive fingerprinting categories. If a user clicks on a link, the referer header will contain the address of the website the user was on when they clicked the link. This is mostly used for analytics, logging or optimized caching, but also used for tracking purposes, as well as leaking information [76]. The

Fingerprintjs2

Your browser fingerprint:

6ea68e53a113d1a7c9accebe26b7eb05

Time took to calculate the fingerprint: 125ms

Detailed information:

```
user_agent = Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/80.0.3987.14
language = nb-NO
color_depth = 24
pixel_ratio = 1
hardware_concurrency = 12
resolution = 1920,1080
available_resolution = 1920,1040
timezone_offset = -120
session_storage = 1
local_storage = 1
indexed_db = 1
open_database = 1
cpu_class = unknown
navigator_platform = Win32
do_not_track = unknown
regular_plugins = Chrome PDF Plugin::Portable Document Format::application/x-google-
chrome-pdf~pdf,Chrome PDF Viewer::
canvas = canvas winding:yes~canvas
fp:
webgl =

adblock = true
has_lied_languages = false
has_lied_resolution = false
has_lied_os = false
has_lied_browser = false
touch_support = 0,false,false
js_fonts = Arial,Arial Black,Arial Narrow,Book Antiqua,Bookman Old
Style,Calibri,Cambria,Cambria Math,Century,C
```

Figure 2.5: Example of information collected by fingerprintjs2. Fingerprint taken from my computer, on the website www.ddzero.net.

URL that is included in the referer header can also contain more information, using the GET technique to append information like search terms and login, and it can even be used to leak user identifiers and e-mail addresses [20]. Hence, if a user visits a website which embeds elements from a third-party tracker, an identifier (for instance a cookie) can be sent to the third-party along with the e-mail address appended in the referer header. An example of how this could look like is demonstrated by Krishnamurthy et al. [84], when a user visits *sports.com* which requests elements from the tracking service *doubleclick.net*:

> *GET http://ad.doubleclick.net/adj/...*
> *Referer: http://submit.SPORTS.com/...?email=jdoe@email.com*
> *Cookie: id=35c192bcfe0000b1...* [84, p. 4]

User-Agent String

The *User-Agent* [123] is a string that contains characteristics about the client's machine. An example of a user agent string (from my computer using Google Chrome) is:

Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.149 Safari/537.36

The components of this string are:

- *Mozilla/5.0*: indicates that the browser is Mozilla-compatible [123].
- *(Windows NT 10.0; Win64; x64)*: describes the native platform the browser is running on [123], in this case Windows.
- *AppleWebKit/537.36*: is the platform the browser is using [123].
- *(KHTML, like Gecko)*: is added on Chrome or Chromium-based engines for compatibility reasons [123].
- *Chrome/80.0.3987.149 Safari/537.36*: describes the browser version, and Safari is added on Chrome or Chromium-based engines for compatibility reasons [123].

With all this information, developers can create more dynamic websites that work well and is tailored to specific browser and operating system details. However, this information can be misused for web tracking purposes and can play a role in the making of a fingerprint [20]. There are two ways a web service can gather clients' user-agent: (i) via the HTTP User-Agent header [123]. This header is automatically sent to the website when a web browser makes a HTTP request to it, or (ii) via JavaScript [46]. Using client-side scripts and the *window.navigator* [129] property a website can collect the user-agent string in this way as well.

Web Beacon

Web beacons [109], also called web bugs, is a stateless tracking technique, but is not used as part of a fingerprint. Web beacons are typically implemented as tiny images, often 1x1 pixels in size, making them invisible to the human eye [130]. This is a known technique used by scammers who send out phishing e-mails, to verify that an e-mail is valid [109]. The beacon will make a request back to the sender if the recipient has opened the mail.

Web beacons have become increasingly popular in web tracking as well, and can be used in the same way, by tracking whether users has opened a web page or not [12]. It also has another usage, which shares similarities to *cookie synchronization*. The beacon, or image, can be used to leak or share sensitive information by being embedded on a website, which makes the user's browser forced to make a request to it. The request to the beacon's URL can include URL parameters, to leak cookie values, or other sensitive information, that can potentially include gender, age, zip code and other demographics [109], which is valuable information to an analytics company.

Canvas Fingerprinting

HTML5 introduced a new element, called <*canvas*>. Within the element's area, graphics can be drawn. There are two main ways to draw these graphics. The first is using the *Canvas API* [21]. This API exposes an interface called *CanvasRenderingContext2D* [22], that allows developers to mainly draw 2D graphics. The second method is using the *WebGL API* [125]. WebGL allows developers to create interactive 2D and 3D graphics, but not all browsers support this API. The *CanvasRenderingContext2D* also includes a method called *fillText()* that can be used to draw text inside the <*canvas*> element. This text can also be further customized using CSS [20].

In 2012, Mowery and Shacham [91] presented a research paper that showed the possibility of using the methods explained above to create unique fingerprints for users. In order to create the fingerprint, data from the <*canvas*> element needs to be collected. This can be done in several ways, but these researchers [91] mainly identified two ways. First, the *CanvasRenderingContext2D* exposes a method called *getImageData()* [23]. This method returns an *ImageData* object, which contains the pixel data from the figure inside the <*canvas*> element [23]. These pixels are represented as RGBA values, as integers [91]. The second method to collect <*canvas*> data, is to use the method *toDataURL()* [73], that is a part of the *HTMLCanvasElement* itself. This method returns a data url, that contains the Base64 encoding of an image from the whole <*canvas*> element [91].

The reason why the data collected from the graphics can be used as fingerprints, is because the way the graphics are drawn differ from browser to browser, and often depends on "*the operating system, installed fonts, graphics card and its drivers, and the browser itself, due to font rasterization, anti-aliasing, smoothing, API implementations, and the physical display*" [20, p. 1488]. The data that is collected is hashed to produce a unique fingerprint for users.

This fingerprinting technique can be used together with other fingerprinting techniques to make the likelihood of uniquely identifying a user even higher, but it's also very strong by itself and can be used as the only fingerprinting technique. Mowery and Shacham [91] stated that this method is fast and consistent, because the same details about the browser and operating system will always produce the same result, and it can be used without users knowledge.

Canvas fingerprinting is a widely used fingerprinting technique. In 2014, Acar et al. [3] performed the first study of real-world canvas fingerprinting practices. In their research they studied the Alexa top 100,000 websites looking for instances of canvas fingerprinting, and found a 5% prevalence, indicating that (at the time) canvas was the most common fingerprinting method ever studied.

The five following sub sections about fingerprinting are categorised in the same way as Bujlow et al. [20] have categorised them.

Network Fingerprinting

Network fingerprinting involves information about the network for users and can be used as part of a fingerprint. One of the easiest methods in this category is observing HTTP messages. From HTTP messages, a website can see the users' global network address [20]. From the *HTTP Accept Header* [70], a website can detect the use of a proxy server. If the proxy is set in the browser, Flash applets can be used to circumvent the proxy, and detect the real IP address of the user [20].

Various network tools exist, for instance the *SpeedOf.Me* API [114], that can gain information about the network of the user, such as download and upload speed, latency, jitter and the user's hostname using JavaScript.

Using open databases or JavaScript with API's, such as *IPinfo* [80], a website can collect information based on the user's IP address. The network relevant information includes the user's hostname and Internet service provider.

Location Fingerprinting

Location fingerprinting involves information about the location (both technical and geographical) of the user [20]. As mentioned in the *network fingerprinting* section, the global network address of users can be observed in HTTP messages. *IPinfo*, mentioned in last section, can also extract information about users' location. This includes the city and country the user resides in, as well as their timezone, postal code and geolocation information provided in coordinates [80].

Device Fingerprinting

Device fingerprinting involves the collection of system information. By combining several of these data points, the likelihood of uniquely identi-

fying devices is high. Acar et al. [2] identified several system attributes that can be used as part of a device fingerprint. These include the device's screen size, versions of installed software and the list of installed fonts. Websites can use JavaScript and the *navigator* [129] property to collect some of this information. A *navigator* [92] object can be retrieved by using *window.navigator*. This object contains information such as the user agent string, list of plugins installed in the browser, platform of the browser, mime-types supported by the browser, languages known by the user and much more [92].

The device fingerprint is generally computed by concatenating several of these values and hashing them. Mayer [88] measured the uniqueness of a device by creating a fingerprint consisting only of information collected by the *navigator* property, specifically information about the screen, plugins and mime-types. He discovered that by concatenating and hashing these values resulted in the ability to uniquely identify over 96% of all the browsers in his tests.

Operating System Instance Fingerprinting

The fingerprinting techniques in this category collects information about the operating system of users. This often includes the architecture and version of the operating system, which both can be collected with Flash and JavaScript [20]. In JavaScript, developers can use *navigator.oscpu* which returns a string containing the users' operating system [92]. The system language in use can be detected with *navigatorLanguage.language* and a list of languages known by the operating system can be detected with *navigatorLanguage.languages* [92].

A list of users' installed fonts can also be collected using JavaScript, as shown in [106]. Users' screen width, height, color depth and pixel depth can be easily collected using Javascript and the *window.screen* property [81]. The screen color and screen resolution can also be collected with Flash [62]. Flash can also detect the audio capabilities of users, as well as if users have allowed access to camera and microphone, if the system supports printing and if read access to hard drives are allowed [62].

Browser Related Fingerprinting

These fingerprinting techniques collects information about users' web browser. A common technique is the user-agent string that was explained earlier in this chapter. The *navigator* property discussed in previous sections can also be used to collect information about the browser. Apart from this, Unger et al. [122] identified CSS and HTML5 features that can be used to collect browser information that can be used as part of the fingerprint, without relying on the user-agent string. They found three methods related to CSS that can be used, namely CSS properties, CSS selectors and CSS filters. Browsers implement these differently, and by looking at the differences between them, a particular browser version can be detected [20]. When it comes to fingerprinting based on HTML5, Unger et al. [122]

found that the implementation details of HTML5 within different browsers varies, and by looking at these details, a browser could be identified. They also found that by looking at the order of HTTP headers in browsers, it's possible to determine the particular browser in use. For example, the order of HTTP headers are opposite in Internet Explorer and Chrome [122].

2.4 Tracking Types

Web tracking is generally performed to keep track on users as they browse a particular website, how long they stay on the site, what they click on and similar actions, but also to follow them around the web and collect information about their browsing behaviour on several websites.

Roesner et al. [101] evaluated the tracking ecosystem and investigated tracking properties, and divided trackers into five different classifications or types. This is valuable research and helps us understand who employs trackers and why, as well as to understand how they work at a higher level. The five following sub sections describes the tracking types that Roesner et al. [101] found.

Third-Party Analytics

Third-party analytics are external analytics tools or engines that offer these type of services to websites that wish to have its traffic analyzed. Google Analytics are among the most popular engines for this purpose. Websites that wish to utilize these services from Google Analytics, receives a library in the form of a script that is embedded on their website. When a user visits the website, the script sets a cookie. Because the script runs in the website's own context, the cookie is a first-party cookie with that website's domain, thereby circumventing third-party cookie protection mechanisms. This means that Google Analytics cannot read this cookie by default. However, analytics engines like Google Analytics, often rely on a form for cookie sharing technique, to leak the cookie back to *google-analytics.com*. Because the cookie is not owned by *google-analytics.com*, users will have different cookie values on different websites, which prevents Google Analytics from tracking the user cross-site [101].

An example of this process is shown in Figure 2.6. The website *site1.com* utilizes these type of services from Google Analytics, and embeds the tracking script on their website. This script sets a cookie, and then makes a request to *google-analytics.com* which contains custom URL parameters, to include the cookie value and send it back home.

Third-Party Advertising

Targeted advertisement is perhaps the biggest usage of web tracking. Multiple advertising companies exist, but Google's Doubleclick network is among the biggest. Websites can host ads provided by, for instance, *doubleclick.net*, and embed these on their website as an image or iframe. When a user visits a website that embeds these ads from *doubleclick.net*, a

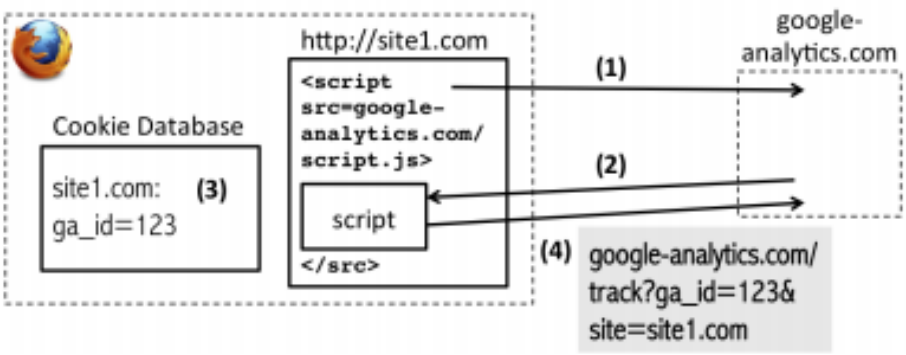


Figure 2.6: Third-party tracking script from Google Analytics loaded in a first-party website. Figure taken from [101].

cookie will be set for that user. The biggest difference between this tracking type and third-party analytics, is that this cookie is a third-party cookie in this context, and is owned by the advertising network. This allows *doubleclick.net* to track users cross-site, as the same cookie is automatically included in all requests made to *doubleclick.net*, even if the user visits another website, as long as the website embeds ads from *doubleclick.net* [101].

An example of this process is shown in Figure 2.7. *site1.com* hosts an ad from *doubleclick.net* and embedded as an iframe. When a user visits *site1.com*, a request is made to *doubleclick.net* to load the ad, which sets a unique cookie for that user. The cookie is owned by *doubleclick.net*.

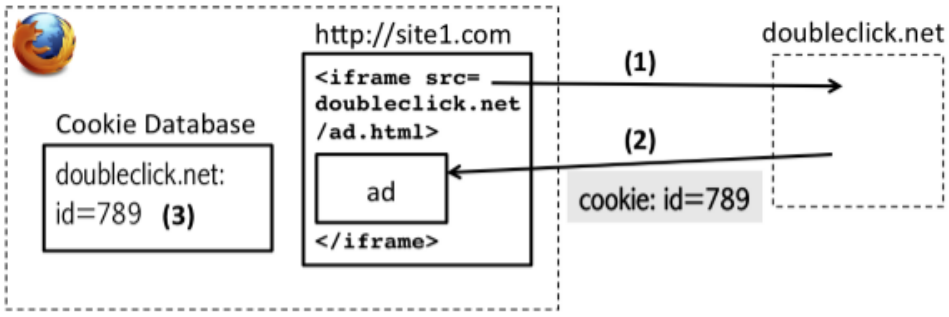


Figure 2.7: Third-party advertising script from doubleclick.net embedded in site1.com. Figure taken from [101].

Third-Party Advertising with Popups

This tracking type is generally aimed at users who have blocked all third-party cookies in their browser settings, or are using privacy tools that limit the use of third-party cookies. Because most cookies set by advertisers are third-party cookies, this method circumvent these countermeasures in a way that forces them into a first-party context. When a user visits a website that is connected to an advertising network, the website will open a popup

window containing an ad. This way the user has been forced to visit this website containing the ad in a first-party context, so the ad can set a "third-party" cookie in a first-party manner [101].

Third-Party Advertising Networks

An advertising network usually consists of a large collection of advertisers. These networks keep track of users as they browse the web, and each will normally have their own unique identifier for users. However, these networks can collaborate and share their unique user identifiers to other advertising networks, to collect more information about them. This can be done if a website embeds an ad from one network, which in turn, makes a request for another advertising network. A form of cookie cookie synchronization/sharing technique is used here, where users' unique identifier is leaked by sending it as a URL parameter with the request.

We can observe how this works in Figure 2.8. *http://site1.com* embeds an ad inside an *iframe* element from *admeld.com*. When a user visits *site1.com*, *admeld.com* sets a unique cookie for that user. After this, *admeld.com* can instruct the browser to make a new request to *turn.com*, and include the cookie value as a URL parameter [101]. The unique identifier (cookie) has now been synchronized between the two advertising networks.

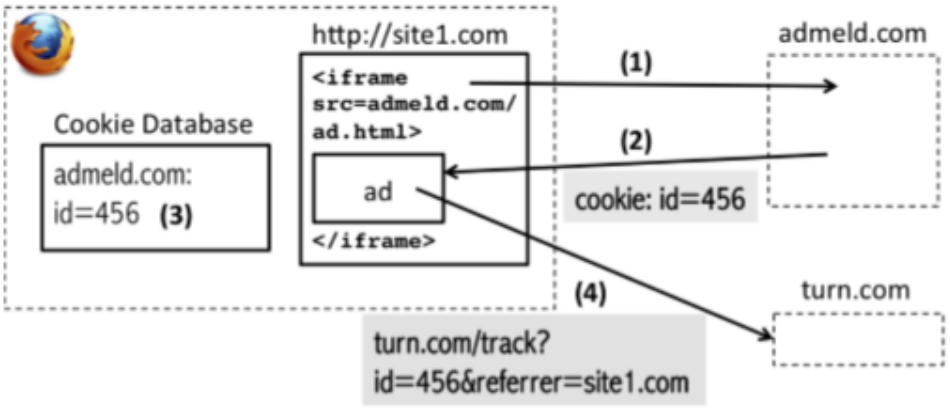


Figure 2.8: Third-party advertising network, admeld.com sets a cookie which is shared with turn.com. Figure taken from [101].

Third-Party Social Widgets

Social widgets generally refer to social media buttons, such as the *Facebook Like* button, *Twitter Share* button and *Google +1* button. The buttons are usually embedded on various websites as third-party web elements. They work by using cookies. When a user visits a website that embed one of these buttons, a request is made to that website, which also sets a cookie for that user. This means that this method can be used as a third-party tracker without the user even interacting with the button. If a user has received a cookie from Facebook in this way, and later visits *facebook.com*

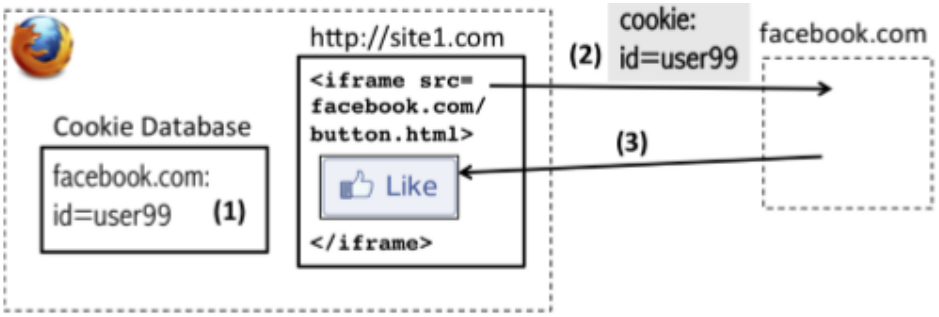


Figure 2.9: Social widget (Facebook "like" button) embedded in site1.com that is used to send a cookie back to facebook.com. Figure taken from [101].

directly, a first-party cookie will be set, and can be merged with the third-party cookie, allowing Facebook to track the user across websites.

An example of this process is shown in Figure 2.9. The website *http://site1.com* embeds a Facebook Like button in form of a script by using an *iframe* element. When the user visits *site1.com*, a request is made to Facebook, which responds with a third-party cookie [101].

Chapter 3

Web Tracking Protection Methods

In this chapter we will look at various protection methods that can be employed to limit or block the web tracking techniques discussed in chapter 2, as well as discuss their pros and cons. In section 3.2 we will look at the state of the art protection tools that are available to Internet users to see how they choose to approach this problem.

3.1 Protection Methods

Private Browsing Mode

The first countermeasure we look at is the *private browsing mode*. Most web browsers generally have a *private browsing* mode, that is implemented with more privacy and security features. Aggarwal et al. [5] studied private browsing mode in Firefox, Safari, Chrome and Internet Explorer (however, Internet Explorer is not relevant anymore). The main purpose behind this mode is not directly tied to web tracking, nevertheless, it does provide some protection. The study looked at different browsing features that set states, where the relevant features for web tracking is history, cookies, HTML5 local storage and the browser's web cache. They did three studies:

- If a user browse the web in public mode, then switches to private mode, are the different states accessible in private mode?
- If a user browse the web in private mode, then switches to public mode, are the different states accessible in public mode?
- If the user browse the web in private mode, are the different states accessible at any point later in the same session?

In their first study [5], they discovered that states set in the four features mentioned above, were not accessible in Firefox and Chrome, but were accessible in Safari.

In the second study [5], they found that the states were not accessible in any of the three browsers.

In the last study [5], they discovered that cookies, HTML5 local storage and the browser's web cache actually were accessible in the three browsers at some point later in the session, but not the history.

This means that using private browsing mode can act as a defence against some web tracking techniques. Optimally, users should commit to only using private browsing mode, but it's not ideal. In situations they really want to not be tracked, they can use the private browsing mode, but remember to close and re-open the browser before going back to the normal public mode. This will prevent websites from setting cookies while in private mode. If the website does have a cookie on the client's machine, at least it won't be sent to the website while in private mode, so they won't be able to track these visits. The evercookie will also be rendered less effective, as the local storage is not accessible.

Using private browsing mode is therefore an easy-to-use method to prevent some form of web tracking.

3.1.1 Stateful Tracking Techniques

In this section, we will look at the stateful tracking techniques explained in chapter 2, and potential countermeasures to each of them.

HTTP Cookies

HTTP cookies have a number of possible countermeasures. Third-party cookies are the most common tracking method, but first-party cookies are also used by analytics engines, like Google Analytics. The most standard way to defend against these cookies is to outright block all of them. This can easily be done in most browsers. For example; Google Chrome has settings to block all third-party cookies, and both Firefox and Brave Browser has settings to block either first- or third-party cookies (or both). However, Roesner et al. [101], found that with these settings on, only Firefox actually blocks both the setting and the reading of cookies. Chrome, Safari and Internet Explorer blocks only the setting of third-party cookies. This means that a website directly visited by the user (e.g. Facebook.com), can set a cookie in a first party context, and when the user later visits another website that embeds trackers from Facebook, the cookie is sent back to Facebook [101]. Using this method will therefore only block third-party cookies if the user never directly visits the domain that set them.

Blocking all cookies will prevent web tracking attempts, but it will also break many websites, as most websites rely on cookies to provide functionality.

Users can also go into their browser settings and clear the cookie storage, for example every time they close the browser. This way they will be tracked throughout a browser session. But on their next session, websites won't be able to link their last visit to the new one, as the cookie values will be different, as long as HTTP cookies are the only way they track the users. However, this is obviously not an optimal solution, but does provide some kind of protection nevertheless.

Cookies can also be removed by developers of web browser extensions. By using the *webRequest* API [28], developers can look into the headers and strip away specific ones, like cookie related headers. This works pretty much the same way as the available settings within the browser itself, but with an extension, it is possible to develop solutions that remove cookies based on some conditions, such as how many times a particular domain sets cookies or the contents of the cookie, which likely will yield a better result than blocking all of them by default.

Cookie Sharing and Synchronization

To be able to share cookie values and/or synchronize them with the back-end, cookies have to be set. The basic countermeasure will be to block all third-party cookies [3], as this will prevent these identifiers being set. But again, this is not an optimal solution.

Another solution is to analyse all HTTP traffic, and inspect all URLs by checking their URL parameters. If a value is seen in a URL parameter that potentially could be a unique value, this can be stripped away from the URL before the request is sent from the browser, however, this solution would likely suffer false positives according to Acar et al. [3].

An improvement of the solution above, is to compare values appended to a URL, as a URL parameter, and check whether this value exist as a cookie value currently stored by the browser. However, this solution would likely suffer from false negatives [3].

If cookies have already been set and can be sent to the corresponding domain, it's impossible to prevent the back-end servers to synchronize the cookies between them.

Apart from these potential countermeasures, it doesn't seem that any other currently exist. Most privacy-related tools will focus on blocking tracking related web resources that set cookies, or just block the cookies themselves.

Flash Cookies

For websites to set Flash cookies, the Flash plug-in needs to be enabled in the web browser. Therefore, potentially the best defence is to disable Flash on all websites by default [20], which will disable Flash cookies from being set at all. However, seeing as some web services require the use of Flash, this is not the most optimal countermeasure.

Flash exposes some settings to users that lets them have some control over the cookies that are set. Users can manually go into the settings and limit the size of the cookies that are set, all the way down to minimum of 10 KB [60]. This will make the cookie just 2.5 times the size of a normal HTTP cookie. Users can also manually enable a setting that prevents these types of cookies from being set in a third-party manner [60], thereby limiting its tracking potential substantially, seeing as most tracking is performed by websites embedding third-party tracking content.

A browser extension called *BetterPrivacy* [13] implemented a counter-

measure to Flash cookies. Instead of intercepting and trying to prevent them from being set, this approach revolves around automatically scanning the folders where the local shared objects are stored, and then deleting them when the browser is closed. However, this still lets the cookie track the user for as long as the browser is opened. Therefore the user should close and re-open the browser regularly. Unfortunately though, it doesn't seem like the extension is supported anymore, however, the countermeasure approach itself is still valid.

Evercookies

The evercookie relies on multiple storage mechanisms to store data. Acar et al. [3] proposed a few countermeasures to evercookies:

- **Clearing storages**: In chapter 2.3.1 I listed some of the possible storages that the evercookie can utilize. Having a program that checks all these storages and simultaneously clear all of them could get rid of the evercookie. However, this solution have a potential weakness. Let's look at an example where two people uses the same computer, but different browsers. The evercookie has stored state in both browsers' storage, as well as in the Flash storage. If one of them clears their storages, the Flash cookie will also be removed from the computer. But when the other person opens their browser, it still exists in that browser storage. This, in turn, can respawn the cookie in the other locations, such as back into the Flash storage. When the first person opens their browser again, the Flash cookie can respawn the previously deleted cookie from the browser storage [3]. This scenario is a little specific, and for most people, might not be a problem.
- **Disabling storages**: The researchers [3] found that users can disable some of the storages altogether, such as the Flash storage, but they also found that localStorage, IndexedDB and canvas cannot be disabled, because they're often used for core functionality. This solution will therefore not provide enough protection.

Bujlow et al. [20] also proposed some countermeasures to evercookies. This mainly revolves around disabling the functionality that the evercookie utilize, such as:

- Partially disallowing cookies and Flash cookies.
- Disabling JavaScript, Flash, Java and Silverlight.
- Frequently clearing all caches, or just browser caches if the above is disabled.

Disabling all these functionalities will severely limit the usability of the web, and are therefore not good solutions.

The developer of evercookie, Kamkar, stated that during his testing, the only current available method to completely stop this technique from functioning was by using *Private Browsing* in Safari [82].

Cache-Based Tracking Techniques

In order to limit the web tracking potential of misusing users' web cache, the browser can be configured in a way to clear the browser cache on every browser exit. This will prevent tracking over time, but not during a particular browser session. Apart from this, it seems that disabling JavaScript is the only effective countermeasure [20].

3.1.2 Stateless Tracking Techniques

Stateless tracking techniques are generally very difficult to defend against. Most data points used as part of a fingerprint discussed in chapter 2 relies on JavaScript and web APIs. Many of these are read-only properties, which means they're not intended to be spoofed (return false values).

In general, the best defence against fingerprinting is to entirely block JavaScript, and potentially Flash and Java [20], as this will prevent fingerprinting scripts to execute and collect information. However, passive fingerprinting from HTTP messages, and collecting information from users' IP address is still possible. Users can use *Tor Browser* or anonymous web proxies as a countermeasure to this [20]. However, blocking Flash, Java and in particular JavaScript, will render most of websites unusable in today's Internet, so better solutions against fingerprinting is needed.

An approach that have been deployed by multiple privacy tools, is to remove functionality and scripts that are known to be performing web tracking. This approach will be explained more in detail in section 3.2. Apart from that, Mozilla released a blog post [59] where they talked about countermeasures to fingerprinting and the future of this field. In their opinion, the primary ways is to either block parties that is involved in fingerprinting (as mentioned above), or change or remove APIs that can be used to collect this information [59]. In the foreseeable future, this will likely involve both.

Another potential approach proposed by researchers focuses on randomization. By adding small random changes, such as noise, to attributes that are used as part of a fingerprint every time a script tries to collect this information, the fingerprint will be different, meaning the tracker might not be able to link recurring visits by the same user. Some of these proposed randomization strategies will be explained below.

Randomizing browser objects. Research conducted by Laperdrix et al. in 2017 [85] focused on an approach to add randomness to attributes that are used as part of a fingerprint. Their aim was to break the stability of fingerprints to prevent websites from being able to link two visits by the same client. Their first solution was implemented within the Firefox browser, where they modified the source code in the *ParseColor* function of the *CanvasRenderingContext2D* class. This function is used to draw the colors on a canvas image. As we saw in chapter 2, the data extracted from the canvas element is unique to different clients. By modifying this function, the researchers were able to add random color modifications. This

means that every time the user visits a website that use canvas fingerprinting, the fingerprint will be different. Secondly, the researchers modified the *AudioContext* API [9], that is used for audio processing. However, it has also been found to play a part in fingerprinting, as details of the audio processing varies between systems. The researchers altered some of the functions that decide the volume of audio processed by this API, by up to 0,001 seconds. This would make the resulting fingerprint hash different.

Lastly, the researchers [85] looked at some of the JavaScript objects, such as *the navigator* and *screen* discussed in chapter 2. In another research paper by Nikiforakis et al. [94], they found that the properties of these objects are listed differently based on the *"browser families, versions of each browser, and, in some cases, among deployments of the same version on different operating systems"* [94, p. 548]. Laperdrx et al. [85] used this information to modify the Firefox source code to change the enumeration order for these objects, and add a random order for each combination. They tested these three solutions on a few common fingerprinters such as *Fingerprintjs2* and *Maxmind*. They used vanilla Firefox as a benchmark, where *Fingerprintjs2* collected the same fingerprint on every browser visit, while with the countermeasures enabled, *Fingerprintjs2* were deceived and tricked into treating every new browser visit as a new fingerprint. They did the same testing on *Maxmind*, with the same result. These results show that even adding small noises to fingerprintable attributes can potentially protect users' privacy from these types of threats.

Randomizing fonts and plugins. The randomization approach was also investigated in another research paper by Nikiforakis et al. [95]. In this research, they focused on randomizing the fonts and plugins of the browser. These attributes are often used as part of fingerprints. The solution they implemented for fonts, was directed at three attributes: *offsetHeight*, *offsetWidth* and *getBoundingClientRect*. These attributes defines the height, width, and size and the position of HTML elements, respectively. The researchers implemented three different policies for how to randomize these attributes, which all were tested. Instead of returning the real values, they would either return zero, a random value between 0 and 100, or $\pm$ 5% noise, depending on the policy in use.

Their implementation for randomizing plugins, defines a probability of hiding certain entries in the plugin list in the browser. Their approach relies on either returning a value of 0 or hiding a certain amount of the plugins in this list.

The researchers [95] tested these solutions on some common fingerprinters, such as *fingerprintjs2*, *BlueCava* and *PetPortal*. The results were that the strongest policies they implemented does indeed prevent all fingerprinting based on font or plugin attributes, while causing minimal breakage of websites on the Alexa top 1,000 websites.

The problem with the randomization approaches above, is that they need to be implemented directly into a browser, or by modifying the source code of a browser, unlike most protection tools that are implemented as browser extensions. The first approach was implemented in Firefox, while

the second was originally implemented in the Chromium browser. In order for these approaches to be successful and usable by the general public, the developers behind these browsers needs to adopt these solutions. Otherwise, a separate browser needs to be created. Recently, some of these randomization approaches were adopted and implemented in a new mainstream browser, the *Brave Browser*, which we will look at in the next section.

Below we will look at the *Tor browser*, which is the most complete browser when it comes to fingerprinting protection, and it implements 30 specific fingerprinting defences [116].

Tor Browser

Tor is a browser that is built on Firefox, but with some modifications, and a more heavy focus on security and privacy, especially when it comes to user anonymity. Their goal is for users to be able to browse the web anonymously. The browser implements many security solutions, and when it comes to web tracking it blocks third-party trackers and ads, as well as clearing cookies and browsing history on each browser exit [118]. These solutions definitely improve user privacy significantly. But more interestingly, is how they defend against fingerprinting. Their aim is to make every Tor user look the same [118]. Below we will look at some of the countermeasures Tor employs.

User-agent and HTTP headers. Tor forces all users to expose the same user-agent and certain HTTP headers, such as *Accept-Language* and *Accept-Charset*, to websites [116]. This will make every Tor user appear the same.

Plugins. Instead of solutions proposed by Nikiforakis et al. that focus on randomization, Tor just disables all plugins, but keeps a setting to enable Flash in case users need it [116]. This approach will prevent fingerprinting based on plugins.

Canvas. Instead of completely blocking the *Canvas* element or introducing noise, their approach detects all read requests to the Canvas API and asks the user for permission before allowing a website to use this API [116]. If the user doesn't allow it, Tor will return an empty image from the canvas functions [3]. According to Acar et al. [3], this is currently the best countermeasure to canvas-based fingerprinting. Another research by Mowery and Shacham [91] also concluded that requiring user approval for canvas-related data is the best countermeasure, because adding noise *could* aid in uniquely identifying a user. This is because the tracker could make several measurements on the same user, and notice that the fingerprint is different each time.

Local network fingerprinting. Tor prevents websites from collecting the local IP address and associated network information to websites [116]. This is done by using proxies and disabling specific APIs.

Fonts. Tor's solution to prevent fingerprinting based on fonts, relies on excluding system fonts altogether. Instead, Tor comes with a predefined list of available fonts [116]. This way, all Tor users will have the same font fingerprint.

Desktop resolution. In order to make users' screen resolution similar,

Tor automatically resizes all new browser windows to a 200x100 pixel multiple, based on the desktop resolution. The window size is also capped at 1000x1000 pixels [116]. In addition, Tor recommends users to not use the browser in full screen mode, to reduce the uniqueness of this fingerprinting method.

Operating system type fingerprinting. Tor identified that at least two HTML5 features are implemented differently based on the operating system and hardware of the client, namely the *Network Connection API* and *Sensor API*. Tor's solution is to disable these APIs in the browser [116].

User-Agent String

As previously discussed, the user-agent string can be used as part of a fingerprint. Seeing as the main purpose of this technique is to provide a better browsing experience, it's difficult to efficiently defend against this without damaging websites. However, there are two main methods to remove or limit its web tracking potential. The positive is that this can be done in a browser extension using JavaScript and the *webRequest* API, instead of having to modify browser source code.

The first method is to simply remove the string altogether [28]. It's automatically sent to websites with HTTP headers when a request is made, but it can be stripped away. This is generally not a good solution though, because the website won't be able to recognize the operating system, browser and versions, and will make the website look worse and not optimized.

The second method revolves around changing the user-agent string. Using the code example provided in [28], one can also change the string before sending it to the website. This idea could work, but the difficulty lies in generating a more generic string that will make the user less unique, but at the same time, enough people need to use the same exact string to gain *crowd anonymity*. This concept means that you need a group of people of sufficient size to use the same thing, and the website won't be able to distinguish between them. This is the idea Tor uses.

Let's look at an example: the user agent-string example provided in chapter 2, *Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.149 Safari/537.36*, is a normal string. One could remove the versions, for instance changing *Chrome/80.0.3987.149* to *Chrome/80*, and similar for the other components. But if there is only one user doing this, they will be just as, if not more, unique. Therefore, to get the most optimal result, a solution like this should be implemented directly into browsers or an extension that many people use.

Changing the value of this string before it's sent to the website also helps prevent the website from collecting the string via JavaScript, as the *navigator.userAgent* returns the value of the string sent with HTTP headers [93].

The web tracking part of the user-agent string might soon be history though, as Google announced in early 2020 a plan to phase out user-agent strings in Chrome [39].

HTTP Referer Header

As seen in chapter 2, this method can be used for both tracking and optimization purposes. It basically has two countermeasures, which are similar to those for the user-agent string.

The entire string can be removed in an extension using JavaScript and the *webRequest* API. This prevents two possible issues: (i) preventing the website to see which website the user was on when they clicked a link redirecting them to the website, and (ii) prevents the possibility of leaking extra information in the header, such as users' e-mail address or other sensitive information.

The above solution will also remove the legit purposes of the header, so instead of removing it completely, it can be changed. Before the header is sent to the website, a program can check its content and strip away anything that is not the exact domain of the website the user was on, to limit its "information leaking" potential, but still letting it perform its optimization tasks. It's therefore a trade-off between privacy and efficient browsing.

3.1.3 Other Efforts to Defend Against Web Tracking

Do Not Track

Do Not Track (DNT) [53] is an optional HTTP header that was originally proposed in 2009 by Soghoian et al. [112]. The header lets users express if they want to be tracked or not. The DNT header can be sent to websites together with other HTTP headers. It has two different values, where *0* means that the user consents to being tracked, while the value *1* means that they don't want to be tracked.

In theory this is great, but there are no legal requirements for websites to honor the DNT system. This means that the DNT signal works on some sites that choose to honor it, although it won't help on others. Much work has been done to try and make the DNT system an industry standard, but it hasn't led to anything conclusive yet.

Removal of Third-Party Cookies

In early 2020, Google Chrome announced that they are going to focus more on privacy in the coming years. Third-party cookies have become so widespread, and threatens user privacy, which results in many users outright blocking all third-party cookies. This prevents most advertisement companies from making money, and at the same time, forces them to implement fingerprinting techniques, which is more difficult to protect against. Chrome is therefore shifting towards an Internet landscape without third-party cookies. By the end of 2022, these cookies will be phased out in Google Chrome [36].

This is a great initiative to improve user privacy, but it remains to be seen if new and improved tracking techniques will emerge, or if this actually ends up in favor of Internet users.

Removal of Flash

In 2017, Adobe announced plans to deprecate Flash [61]. This is scheduled to happen by the end of 2020. This means that content that relies on Flash must migrate to other technologies. For privacy, it means that Flash cookies will no longer be a threat. Additionally, the evercookie will be left with less available storage mechanisms.

Silverlight End of Support

Silverlight provides storage mechanisms that the evercookie can utilize. Microsoft announced end of support for Silverlight, which is scheduled to happen in October 2021 [108]. This will leave the evercookie with even less available storage mechanisms. The evercookie can replicate cross-browser if Flash, Silverlight or a few Java mechanisms are available, so without Flash and Silverlight, this ability will be limited.

3.2 State of the Art Tools

In the last decade we have seen a rise in various privacy tools being developed to be able to effectively block web tracking. The state of the art protection tools we have today is *Ghostery* [66], *Disconnect* [52], *Firefox Content Blocking* (only available on Firefox) [58], *Privacy Badger* [99] and a new browser called *Brave* [16]. These tools and browsers are the most popular among Internet users as well as the most mentioned in research papers and studies. However, other privacy tools also exist, both those developed by researchers, but also those developed by the community. The tools we will look at share the same goal; to block web trackers, preferably without breaking website functionality. They also share a secondary goal, to speed up website load time. Because there exists so many different tracking mechanisms, most tools focuses on a more general approach to block trackers instead of focusing on countermeasures that only applies to each one. They are often divided into a *blacklist-based approach* or an *algorithmic approach*.

In this section we will explore these tools to see how they deal with web trackers and what kind of customization options they provide to their users. We will also look at how easy they are to use for internet users who might not have any prior knowledge of web tracking and Internet in general.

Blacklist-Based Approach

Most of the tools we will look at below relies on some form of blacklist or ruleset to determine what to block. There are many publicly available lists that tools can use. *EasyList* [55] is the most known and is maintained by a handful of community members. Many other lists can be found here [11]. EasyList maintains multiple lists that are used by various ad blockers and privacy tools;

- *EasyList*, a list that removes ads, empty web frames resulting from blocked ads, images and objects [55].
- *EasyPrivacy*, a list that focuses on removing all web tracking, including web bugs, tracking scripts and information collectors [55].
- *EasyList Cookie List*, a list that removes cookie banners, GDPR overlay windows and some other privacy-related notices [55].
- *Fanboy's Social Blocking List*, a list that removes social widgets (e.g. Facebook like button and Twitter share buttons) that can be used to track users [55].
- *Fanboy's Annoyance List*, a list that blocks social media content and other annoyances, such as pop-up windows [55].

The various blacklists are structured differently, but they all contain something that should be blocked. For example, the EasyList contains thousands of entries that are one of these rules:

- *Network rules*, that are either a URL or parts of a URL that is a known tracker and should be blocked [17].
- *Element rules*, that specifies page elements to hide [17].
- *Exception rules*, that specifies exceptions to a rule; even if the resource should be blocked, it won't be blocked if it falls under this rule [17].

An example of some of entries in the EasyPrivacy list is;

```
&trackingserver=
-analitycs/ga.
google-analytics
track-re01.com
```

If we look at the third rule, for instance, this should match any URL where *google-analytics* is included, such as *www.example.com/google-analytics* and cancel HTTP requests to this website. Browser extension can utilize a functionality provided by Chrome called *Match Patterns* [47] to match any URL pattern that contains this rule.

Disconnect, a privacy tool we will look at in section 3.2.2, have their own open-source list that can be found on their GitHub page [48]. These are structured differently than EasyList and only contains domain names of known domains that are associated with web tracking.

The blacklist-based approach works thanks to the *webRequest* API [28]. This lets developers modify or cancel HTTP requests before any more information is exchanged with the website. This approach is therefore generally safe. Lets assume a website employs multiple web tracking techniques. If a privacy tool focuses on detecting and blocking web tracking techniques one by one, and actually fails on properly blocking one

of them, the website could still be able to track the user. When the request to the URL itself is cancelled, no information is exchanged and no script can run to collect information about the user.

These approaches generally have a few weaknesses. Only known trackers can be classified as such and added to the blocking lists. Trackers can change their URLs, which means the list has to be updated to reflect this change [65]. This approach will therefore always lag behind the trackers, but if it's updated regularly it shouldn't take long before the new tracker is added to the blocking list.

Secondly, developers sometimes add tracking code (either intentionally or unintentionally) to originally non-tracking scripts [128]. In this case, blacklists might include this script and thereby break website functionality. If the developers later change the script to not collect any information that is regarded as a tracking attempt, the script will likely stay in the blacklist. Therefore every entry in the list should be assessed every once in a while to confirm that it's still used for tracking purposes [131].

Another weakness identified by Yu et al. [131] is the fact that most blocking lists contain (at least partially) second-level domains (e.g. *example* in case of *example.com*). In this case the whole domain will be blocked. Tracking scripts in those domains may only be implemented in specific sub-domains or in a specific URL path. The problem comes if this domain also provide important functionality to a website. This will then cause the website to break in certain areas, and users might whitelist (if this option is implemented in the defence tool) the whole domain, which renders the blocking list useless for that domain.

Lastly, a problem identified in a study conducted by Brave [17] that mainly applies to huge lists, such as the community maintained EasyList, is the fact that it's growing steadily over time, as more rules are added than removed. This leads to a build-up of potentially stale rules (rules that are old and either doesn't block anything anymore, or block domains that no longer poses a privacy threat), and no effective way of identifying and removing these. This eventually leads to an increased size of the list and over time, a bigger performance cost of running it [17].

Algorithmic Approach

An alternative to the blacklist-based approach is an algorithmic one. Some tools, like Privacy Badger that we will look at in section 3.2.4, implements a series of algorithms to try and identify and block trackers without relying on a blacklist. These solutions can differ a lot between the tools, and will be further elaborated on in the appropriate sections.

3.2.1 Ghostery

Ghostery calls itself a *Privacy Ad Blocker*. As mentioned above, it's goals are to limit (or ideally block) all web tracking, ads and to speed up website loading [66]. As of August 2016, if we exclude the pure ad blockers like *Adblock Plus* [4], Ghostery has the biggest userbase with a combined

3,686,040 users on Firefox and Chrome. Ghostery first came out in 2010, but was bought by the German company *Cliqz International GmbH* in 2017. It is implemented as a web browser extension, mostly using JavaScript. It is available in most browsers today, including Firefox, Chrome, Opera, Safari and Edge. It is also available on mobile for iOS and Android.

Ghostery has an extensive user interface with many options and configuration settings. When first installed, Ghostery will present which third-party trackers it detects, but not automatically block them. The user has to configure it themselves, which seems to be too much to ask for the average Internet user. Leon et al. [87] studied the usability of several privacy protection tools, including Ghostery, and found that two out of the five Ghostery users thought they had enabled the extension to block trackers, while in reality, they had not.

Ghostery's user interface shows how many trackers are blocked and/or modified and the page load time. It also has a *Detailed View* tab where users can see detailed information about the trackers Ghostery has detected. Users can click the *Trust Site* button to whitelist a page, which will remove all anti-tracking mechanisms for that website or *Restrict Site* to blacklist a page, which will enable all anti-tracking mechanisms for that website.

Ghostery mainly works by using a blacklist-based approach to stop trackers. This means that it relies on a list that contains the urls (or parts of urls) that is known to perform web tracking. As of April 2020, this library consists of more than 4500 trackers from more than 2600 companies [63]. When an HTTP request is made to a website, Ghostery monitors the call and matches them with the entries in the blacklist. If there is a match, the call will be cancelled before any more information can be exchanged [71].

Generally all blacklist-based approaches relies on how good the blacklist is, which also applies to Ghostery. Ghostery relies on a centralized approach to create the blocking rules [89]. This means that the company behind the tool maintain and curate the blocking rules. Ghostery's blacklist is heavily formatted and copyrighted, so no one else can use it.

However, when Ghostery was acquired by Cliqz, some of the privacy mechanisms that Cliqz use was integrated into Ghostery. Cliqz [40] is a browser and search engine with privacy features built into it. Cliqz used an algorithmic approach to protect against web tracking, and this is the solution that was integrated into Ghostery as an optional feature to supplement the blacklist-based approach. This anti-tracking system [115] relies on observing HTTP requests and looking for different conditions. The system is divided into two subsystems;

1. *Cookie Protection*. This method checks whether the user interacts with a particular web resource that, in turn, relies on third-party cookies. If the user doesn't interact with it, the cookies associated with this resource is blocked. Otherwise, the resource is temporarily whitelisted, to allow the use of cookies to make sure the functionality is usable, and the page doesn't break. [115].

2. *Unsafe Data Removal*. This system attempts to evaluate whether a

request contains data that is "unsafe" or "safe". This is a difficult task. If they find that the data sent contains uniquely identifying data for that user, they will remove the data before the request is sent to the website. Their algorithm for this solution has four steps [115];

"1. *Analyse the URL, headers and postdata of the request.*
2. *Tokenise this data into key-value pairs.*
3. *Evaluate the safeness of each key-value pair.*
4. *If there are unsafe values, remove the data from the request."* [115]

Ghostery maintains a type of library that contains values that cannot be unique identifiers, which is continually updated automatically when Ghostery users browse the web. The evaluation of the safeness of key-value pairs is done by detecting values that cannot be unique identifiers, and removing all other data. This solution protects users right away, and in most cases will remove uniquely identifying information from the requests [115].

Ghostery mainly has three different anti-tracking mechanisms to further customize the tool;

- *Enhanced Anti-Tracking*
  - This is the name of the algorithmic anti-tracking mechanism adopted from Cliqz and can be turned on or off.
- *Enhanced Ad Blocking*
  - Seeing as Ghostery is a privacy ad blocker, enabling this feature will also block ads.
- *Smart Blocking*
  - This feature will try to optimize page performance while still blocking trackers. This involves automatically blocking slow or non-secure trackers, but allow trackers that may break on popular websites when blocked [64].

Merzdovnik et al. [89] performed a large-scale study of tracker-blocking tools in 2017. Ghostery was one of the tools they analyzed. They ran a crawler on 191,492 websites and found that Ghostery performed better than both Disconnect and Privacy Badger, with a success on 179,068 out of the 191,492 websites.

Ghostery's aim with combining a blacklist-based approach and an algorithmic one, aims at reducing website-breakage by not having a too strict blacklist, and instead of having the algorithms block data from being sent, they only strip away uniquely identifying information. This is therefore a compromise; to keep website functionality intact while still protecting user privacy.

3.2.2 Disconnect

Disconnect's mission is to make it easier for people to exercise their right to privacy [52]. Their goal is to block trackers and speed up website loading. Disconnect is implemented as a web browser extension, like Ghostery, and is available on most browsers, like Chrome, Firefox, Safari and Opera and more. It is also available on mobile. As of August 2016, Disconnect had a total of 1,062,870 users on Firefox and Chrome, second to Ghostery [89].

Disconnect's user interface is quite minimalistic with fewer configuration settings than Ghostery. However, Disconnect is easier to use for the average Internet user because no configuration is needed, and most trackers are blocked by default. On the top of the dropdown menu there are buttons to block or unblock Facebook, Google and Twitter for easy configuration for those popular websites. They also have a button for whitelisting websites which will turn off blocking mechanisms for that website, and blacklisting button for blocking all requests to that website. Further, Disconnect divids trackers into four categories; *advertising*, *analytics*, *social* and *content*. Clicking any of these, users can see what company is trying to track them and how many requests they make. There is also a checkbox used for easy blocking or unblocking.

Disconnect mainly relies on a similar blacklist-based approach as Ghostery, but their internal ruleset is different. Disconnect's lists are open source and can be found on their GitHub page [48]. Disconnect's database of trackers are maintained by crawling popular websites and looking for third-party requests, which are then categorized as one of the four categories mentioned above [111].

This is the underlying way Disconnect block trackers, but the free version of Disconnect consists of two features;

- *Private Browsing*
  - This feature blocks invisible websites that attempts to track user's search and browsing history, as well as visualizing this process [20].
- *Private Search*
  - This feature lets users use search engines without allowing them to track their searches and can be found here [51].

Disconnect has an extensive premium version [50] which comes with more features for better privacy. This includes an optional full VPN which can mask IP addresses and encrypt all data.

The survey by Merzdovnik et al. [89] (mentioned under section 3.1.1 Ghostery) also included Disconnect. It performed well, close to Ghostery's results, with a success on 176,659 out of the 191,492 websites.

3.2.3 Firefox Content Blocking

Mozilla have had a big focus on privacy lately, and improved their privacy options considerably by implementing a set of settings they call *Con-*

tent Blocking [58]. The big difference to other privacy tools, is that content blocking is built into the Firefox browser, and is not some extension that has to be downloaded separately. Firefox Content Blocking is basically using the same underlying methods to block trackers as the previous discussed tools. They rely on monitoring HTTP traffic, and requests to known third-party tracking domains are cancelled if they match an entry in their blacklists [83]. Firefox relies on blacklists provided by Disconnect, which makes these two tools somewhat similar [58]. Disconnect has what Mozilla calls a level 1 and level 2 list. Both of these can be enabled by the user in Firefox Content Blocking settings to block more trackers.

Firefox Content Blocking also provide defence against third-party cookies, however, it's very strict. Users can choose to block all third-party cookies and/or all first-party cookies.

In January 2020 Mozilla wrote a blog post where they announced a collaboration with Disconnect, mainly in order to provide better protection against fingerprinters [59]. This collaboration involves letting Firefox use Disconnect's blacklist for fingerprinters, and Mozilla helps them to classify domains as fingerprinters. This is done by crawling the web and looking for scripts with fingerprinting signatures. If we look at Disconnects GitHub [49] we can see that the fingerprinting script mentioned in chapter 2 figure 2.3, called *fingerprintjs2*, is used by many of the known fingerprinters.

Firefox classifies trackers as cookies, fingerprinters and cryptominers so the user can easily choose what they want to block. Content Blocking has three different settings [58];

- *Standard*
  - Standard is the basic setting and the default if the user doesn't customize it. Standard mode blocks social media trackers (e.g. Facebook like button and Twitter share button), third-party cookies, cryptominers and fingerprinters which are known, but only in *private browsing window* [58].
- *Strict*
  - Strict blocks everything that the standard mode blocks, but in every browsing window. However, they warn that it could affect functionality (basically block legit scripts or cookies) on some websites [58].
- *Custom*
  - Custom mode lets the user choose what they want to block, based on the classifications. This means the user can choose to only block fingerprinters and cryptominers, for instance. They can also choose which browsing windows they want to block trackers in, and also if they want to block all third-party or first-party cookies [58].

Firefox Content Blocking also lets the user send a *Do Not Track (DNT)* signal to known trackers or all websites to let the website know that the user does not wish to be tracked.

3.2.4 Privacy Badger

Privacy Badger [99] is another tool that can be installed in browsers. It is implemented as a browser extension like Ghostery and Disconnect, and is available on Chrome, Firefox, Opera and Firefox for Android. It is developed by the Electronic Frontier Foundation (EFF) and first released in 2014. Although EFF states that they like Ghostery and Disconnect (and similar products), they weren't exactly what they were looking for [99]. They wanted a program that didn't rely on blacklists, but rather an algorithmic approach, that would analyze web resources before determining to block it or not. [99].

Privacy Badger works by observing all third-party requests, whether they are used to load images, ads or other necessary functionality, and logs all of this, along with the website the user directly visits (the first-party). If they find that third-party requests are sent to the same domains when visiting different first-party websites three times, they will be blocked [99]. EFF states that Privacy Badger looks for tracking techniques such as *"uniquely identifying cookies, local storage "supercookies", first to third party cookie sharing via image pixels, and canvas fingerprinting."* [99].

They also state that if third-party HTTP requests goes out to what they recognize as important website functionality (such as images, embedded maps or stylesheets) they will allow the connection to those third-parties, but remove cookies and referer headers from being sent [99].

Privacy Badger is the first tool we look at that tries to detect and prevent fingerprinting (without using a blacklist). Most fingerprinting techniques are very difficult to block, and EFF states that that is an ongoing project. The fingerprinting they try to detect is canvas fingerprinting, that we looked at in section 2.3.2. Privacy Badger looks for the use of the HTML canvas element within websites, and treat these as tracking attempts [107].

When it comes to the local storage "supercookies", the way Privacy Badger treats this as a tracking attempt is by looking at how many read or writes the script makes to a third-party local storage [107].

It is generally difficult to distinguish if a cookie is used for tracking or legit purposes, but Privacy Badger tries a method where they look for more information that is exchanged with the cookie. If they think the cookie contain enough information to uniquely identify a user, it will treat it as a tracking cookie [107].

Cookie sharing is a technique that was explained in chapter 2 under *Web Beacon*. Privacy Badger tries to detect this technique by running three checks for every third-party request [107]:

1. As we saw in chapter 2, web beacons are often used to share cookie values or other unique identifiers, where the values can be included in a request URL. Therefore, Privacy Badger first look for requests

that originate from a small image (often 1x1 pixel) [107].

2. The unique identifiers are included in the URL as a parameter, so the second check is to see if the image request URL contains extra information [107].

3. Lastly, it checks for first-party cookies, and compare parts of these cookies (at least 8 characters) with the query arguments in the URL parameter [107].

If all three conditions are true for a request, it's treated as a cookie sharing attempt and this information will be logged. Potentially, Privacy Badger will take action.

In 2012, Roesner et al. [101] developed and implemented an extension called *ShareMeNot* [105]. The goal of this extension was to defend against tracking via social media widgets (see chapter 2.4). By default, these widgets can act as trackers using third-party cookies, even when a user doesn't interact with it. The extension worked by removing cookies in the requests made by the widgets, unless the user actually clicks it. This is different than other solutions which outright remove the buttons from websites. In 2014, this extension was no longer supported as a stand-alone extension, and its functionality was incorporated into Privacy Badger [20].

Privacy Badger comes fully configured out of the box, so it's very easy for amateurs to use. It has a simple user interface, which shows how many potential trackers it has detected. We're also shown all trackers in a list, along with a slider. The user can move these sliders to adjust how Privacy Badger deals with each tracker. If the slider is all the way to the left, the tracker is completely blocked. If the slider is in the middle, Privacy Badger treats it as a necessary element for website functionality, and will allow it, but block cookie and referer headers. If the slider is to the right, the tracker will be allowed to execute. The user interface also shows the user which third-party elements it finds that might be a tracker, but has not yet learned to block. If the user knows this is a tracker, they can drag the slider to block the element before Privacy Badger learns it is a tracker.

When using Privacy Badger for the first time, it won't know what to block so it lets all trackers execute. This is definitely a weakness, although, over time it will learn what to block. This is an interesting alternative to the blacklist-based approaches, but according to the study performed by Merzdovnik et al. [89] Privacy Badger suffers from a lot of false positives. They trained the program with the Alexa top 1,000 and then ran an evaluation. As we saw with Ghostery and Disconnect, they performed quite well. However, Privacy Badger caused 28.56% of all websites to fail to load (they experienced time-outs). This is the weakness of an algorithmic approach. The study concluded that this method showed promising results, and future research should focus on better methods to detect whether blocking a certain element will break functionality of websites.

3.2.5 Brave Browser

Brave is developed by *Brave Software* and originally came out in 2016, but with few features, and plans for privacy protecting features in the future. It wasn't until 2019 that Brave really became known, when they had implemented new techniques to protect user privacy. The browser is based on the Chromium web browser [37] and is available on Windows, macOS, Linux, Android and iOS. Because privacy was a concern for the developers of Brave, they had the chance to develop Brave with a privacy-focused mindset from the beginning. This lets developers incorporate protecting measures more efficiently than having a browser extension to do it. Brave comes fully configured with privacy settings on when first installed, which makes it an easy and good browser to use for amateurs.

Along with all the other privacy tools we have looked at, Brave advertises a faster load time of web pages, specifically 3x to 6x faster than Chrome and Firefox [18]. Brave comes with multiple security and privacy features. The features relevant to privacy are [18];

- *Ad blocking*
- *Fingerprinting prevention*
- *Cookie control*
- *HTTPS upgrading*
- *Block scripts*
- *Send "Do not track" with browsing requests*

Brave, like the other tools we have looked at, relies on some form of blacklists. However, they try to improve on this solution and add more techniques on top of that. A known problem with blocking trackers is that websites, or at least some functionality on the site, may break. This is because JavaScript code that performs the tracking might be "baked" into the main functionalities of a website, either by accident or intent by the developers to make privacy tools choose between allowing the user to visit the website (and also be tracked) or to break the website [128]. Brave attempts to implement solutions to remove this concern altogether, with a technique they call *resource replacement* [128]. The technique works by replacing code that makes a request to a third-party tracker with customized code that still makes a network request, but not to the tracking-code [128].

For ad blocking Brave uses blacklists (or filter lists as they call it) [19]. In their lists there are network and cosmetic rules. The network rules basically makes up the blacklist; they specify which URLs should be blocked. A URL can include a third-party tracking script (for example in the form of an ad) by, for instance, using the *iframe* HTML element. When this is blocked, the area where the ad should have been is now empty white space. Brave attempts to remove this space by using cosmetic rules to hide these empty areas to make the website look better. [126]. One of the ways Brave achieves

the fast page loading times is due to their ad blocking optimization. They implemented a new engine in Rust, with a new algorithm that optimizes the use of the blacklists to achieve a 69x faster page load than the current engine [19].

For their fingerprinting prevention, Brave removes the widespread canvas and WebGL APIs from third parties by default [127]. This prevents third-parties from fingerprinting users with these methods, but will also break legit website functionality that relies on third-party scripts that utilize these methods. Brave recently further improved on their fingerprinting prevention by implementing a technique they call *fingerprint randomization* [127]. This is based on the work done by Nikiforakis et al [95]. and Laperdrix et al. [85] that was explained in section 3.1.2. According to Brave [127], this is the first time approaches based on randomization have been implemented in a mainstream browser. Brave's implementation of this technique attempts to remove the possibility for websites to link recurring visits by randomizing some of the values fingerprinters will gather. By adding subtle randomization to some of these values on every new browser session, the client will look unique on subsequent visits to the same website [127]. These randomization solutions are currently available in *Brave Nightly*, which is a version intended for testing. The plan is to implement these randomization techniques in the main browser version by the end of 2020.

The cookie control feature lets the user choose which types of cookies to allow. By default, Brave allows all first-party cookies and blocks all third-party cookies. This feature is implemented with three options that can be chosen on a site-by-site basis: [15]:

- *"Cross-site cookies blocked: Accepts 1st party cookies and blocks any others on the site.*
- *Cookies blocked: Blocks all cookies, both 1st and 3rd party on the site.*
- *All cookies allowed: Accepts both 1st and 3rd party cookies on the site."* [15]

The HTTPS upgrading feature tries to use HTTPS on every website if it knows it's supporting it. This will force an encrypted connection [15]. Some websites may default to HTTP (unencrypted connections) but this feature fixes that problem. The feature is based on the tool *HTTPS Everywhere* [77] that is a collaboration project between the Electronic Frontier Foundation (EFF) and The Tor Project.

The block scripts feature [15] is an advanced feature which users has to be careful with using. By default, Brave allows websites to run JavaScript. With this feature enabled, JavaScript will be blocked, but users can choose to fine-tune this setting and specify exactly which scripts to allow or block. Most of the web relies on JavaScript today, so disabling scripts to run will break websites.

The send "Do not track" signal feature [15] can be turned on or off, but won't impact the user experience in any way. It's sent along with other HTTP headers to websites to let the client express their wish to be tracked

or not. Some websites choose to honor this signal, but there are no requirement for them to do so.

Not a lot of research has been done on Brave because the browser is fairly new, and most of the privacy features have been implemented recently. But Leith et al. [86] conducted a study in February 2020 on browser privacy, by assessing the privacy risks regarding back-end data exchange. They evaluated the back-end services used by Brave, Chrome, Firefox, Safari, Edge and Yandex with a focus on privacy. They found that Brave (with default settings) did not transfer any persistent data to back-end services when the browser was closed and re-opened, and they did not find any use of identifiers which allows tracking of IP address over time. Brave was the only browser where both of these criteria were true. Based on this privacy perspective, they concluded that Brave has the best privacy, followed by Chrome, Firefox and Safari which shared second place. Because Brave was developed with the intent of having better privacy for users and all of these privacy features "baked" into the browser, it comes as no surprise that they beat the other browsers where privacy was an afterthought.

Chapter 4

Design and Implementation

This chapter describes the design and implementation of a system to protect user privacy against web tracking threats.

4.1 Architecture

This system was developed with two goals in mind:

- **The main goal** is to block as many web trackers as possible.
- **The secondary goal** is to prevent breaking important website functionality. This is a trade-off between protecting user privacy and usability.

Similar programs used to protect against web tracking has another important goal that they also focus on, which is to speed up web page load time. This has not been a goal for me, because it's not a part of the project, and it is not intended to be used by the public in its current state.

Web Browser Extension

Most web browsers today are extensible, which means they can be extended to perform more tasks than originally implemented in the browser. Developers are free to develop programs (browser extensions) to further customize the browser in many ways. Some examples include changing color, fonts and text on websites, measure load time of websites, block certain web elements from loading and much more [25].

This system is implemented as a web browser extension. This choice was made because in order to block tracking scripts and other tracking techniques, we need to easily get access to browser components and communication between the browser and the Internet. A browser extension can roughly be looked at as a middle-man between the browser and the Internet. Using the appropriate APIs, an extension can inspect communication between the browser and Internet and take appropriate action to prevent undesired data being received by the browser, and thereby reaching the client's computer.

The extension is developed for the Google Chrome browser and tested for version *81.0.4044.138*. Extensions work similarly between most browsers, so porting it to another is not too much work. It was developed for Chrome due to the simple fact that it appeared slightly easier to learn than for other browsers such as Firefox.

Before delving into how the system works, it is important to understand the basic concepts of how browser extensions work. Extensions consists of multiple components [30]:

- **Manifest:** The manifest file is a JSON-formatted file named *manifest.json*. Every extension must have this file. It contains important information regarding the extension, such as defining files as background, content or popup scripts, as well as to give the extension permissions to use different APIs [34].
- **Background scripts:** Background scripts are JavaScript files that contain source code. These scripts are executed when the extension is turned on (or the browser is opened) and then goes idle until some of the code is needed. Only background scripts can do API calls (except for calls to the storage API), so code from the background is usually fired when a content or popup script sends a message and asks for it, or if the background script contains some code related to network requests. Those will execute automatically when a network request is made [33].
- **Content scripts:** Content scripts are also JavaScript files that contain source code. These scripts run in the context of a web page. This means that every time a web page is loaded or refreshed, these will run. They make use of the Document Object Model (DOM) to read and change details about web pages [31].
- **Popup scripts:** Popup scripts are also JavaScript files that contain source code. These scripts are used to display information in the popup window of the extension, and should open when the user clicks the extension icon in the web browser [32].

These are the main components of an extension, but the popup.js file usually have an associated HTML and CSS file to customize the popup window. Some extensions can also have background scripts as HTML instead of JavaScript. An extension can also utilize multiple scripts of the same type to divide certain parts of the program in different files. As mentioned, background scripts are the only scripts that can do API calls. If a content or popup script wants to do that, they have to use the *message passing* functionality [35] to send messages to notify the background scripts. The *runtime.sendMessage* or *tabs.sendMessage* lets scripts send one-time messages (small variations in the code depending on if you send from a background to a content or vice versa) to another script. Long-lived connections are also possible, using *runtime.connect* or *tabs.connect*

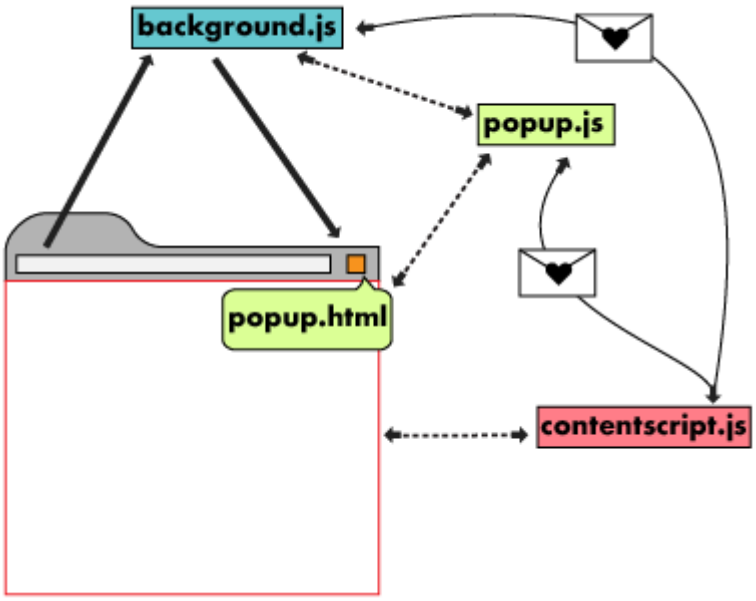


Figure 4.1: Overview of browser extension components. Figure taken from [30].

a channel can be opened between two scripts to let them communicate several messages between them. There is also one more way content and popup scripts can communicate with background scripts, by storing values in local storage using the *chrome.storage* API [29], which lets the background script read these values later. The storage API is the only API accessible by content and popup scripts.

In Figure 4.1 we can see an overview of the components in an extension, where the envelope icon represents how the scripts can communicate using message passing.

Programming Language

Browser extensions are built on web technologies, like HTML, CSS and JavaScript [25]. This means the programming language used is in this extension is JavaScript. JavaScript is a client-side language, which means the script is executed on the client side. The language is used for most of the web, to create websites and much more. In chapter 2, we also saw that JavaScript is the preferred language used for tracking scripts, because of this reason, and also because it has many available web APIs. Seeing as JavaScript is the only available language to create browser extensions with, it was the default choice of language. Some HTML is also used, specifically for the user interface.

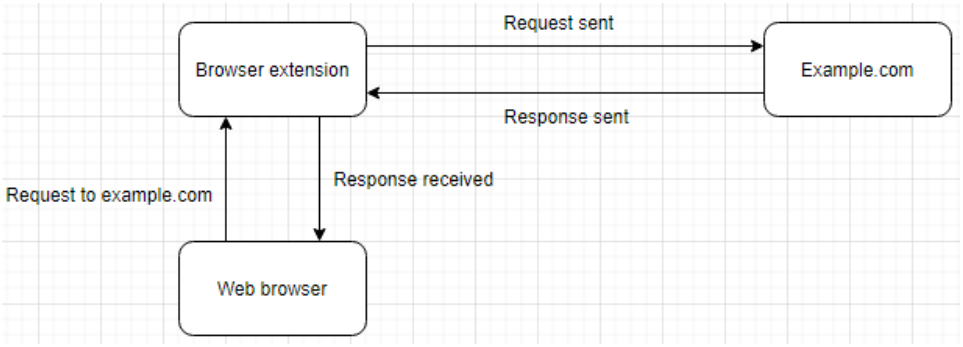


Figure 4.2: Communication flow between a browser and a website when the browser extensions intercept the request, but lets it go through.

Communication Flow

The normal communication flow between a browser and a website, where the browser is using the extension that intercept the communication between them is observed in Figure 4.2. The browser (client) and the website (server) communicates with HTTP messages, and these specific messages are the ones we want to observe and inspect.

We see that the browser makes an HTTP request to *example.com*, which is temporarily paused within the extension before it determines how to deal with it. If the request is deemed okay, the extension will let it go through. A response is sent back from *example.com*, and again, paused within the extension. If the response is also deemed okay, it will go through and reach the web browser.

4.2 Design

The task of this project is to protect user privacy against web trackers. In order to effectively do this, I need to implement a program that blocks a wide array of tracking techniques. In chapter 2 we saw that there is a vast amount of tracking techniques available. I discovered that HTTP cookies are the most widespread method. Supercookies, such as Flash cookies, are not that widespread. I also discovered a wide array of fingerprinting techniques in use, and in chapter 3, I found out that most of these have no or limited effective defences against them. With these discoveries in mind, I decided to implement two main protection methods which will be explained below.

4.2.1 Method 1. A Blacklist-Based Approach

A blacklist (or blocking list) contains the signature of known trackers. This can be domain names, full URLs, parts of URLs, paths that applies to multiple domains, IP addresses and more. These signatures are often related to scripts or images, where the scripts can contain various

fingerprinting techniques or the use of more advanced cookie techniques (such as the evercookie, that relies on a JavaScript API), while the images can be used as advertisement banners to set cookies, or as pixels to share or leak sensitive information, such as cookie values.

The idea with using a blacklist is to prevent/block all forms of web tracking from a particular resource or URL, but it also blocks all other content from that particular resource from being loaded into the browser. This have both positive and negative impacts. The positive is that if a tracker use both a known and an unknown tracking technique, both will be blocked. The negative is that if the same script is mainly used to provide important website functionality, and a side-effect of the script is to collect information, or the developers intentionally added tracking code into the script, the creators of the blacklist needs to evaluate whether to add this into the blacklist or not. This could be a difficult decision, and is a trade-off: privacy vs. usability. Other strengths and weaknesses with this approach were discussed in section 3.2.

There are many available blacklists to use. I decided to implement two different ones. The *EasyPrivacy* list provided by EasyList, and Disconnect's lists which are also used by Firefox and Microsoft Edge. There is some overlap between the lists, but it will provide more protection using both compared to just one of them. EasyPrivacy contains around 16,000 entries (as of April 2020), while Disconnect's list contains roughly 6,000 entries, but it's hard to count them. EasyPrivacy includes full URLs, parts of URLs, paths that can apply to multiple domains and IP addresses. Disconnect's lists contains only domains (e.g. example.com). Their *services.json* file contains domains that acts as third-party trackers on some website, but provide core functionality to other websites. An example from this list is:

```
"Twitter": {
  "https://twitter.com/": [
    "ads-twitter.com",
    "tweetdeck.com",
    "twimg.com",
    "twitter.com",
    "twitter.jp"
  ]
}
```

In this case, these domains should be registered as trackers, and third-party requests to these should be cancelled. However, the five domains specified below *https://twitter.com/* provides core functionality to *twitter.com*, and should therefore be whitelisted (and not blocked) when a user directly visits *twitter.com*.

Disconnect's *entities.json* file also contains domains that should be registered as trackers, and have third-party requests to them cancelled. However, it specifies organizations, and other domains that are part of the same organization below it, which should be whitelisted for that specific organization [119]. An example is:

```
"365Media": {
    "properties": [
        "aggregateintelligence.com"
    ],
    "resources": [
        "365media.com",
        "aggregateintelligence.com"
    ]
}
```

In this example, *365Media* is an organization that owns both *365media.com* and *aggregateintelligence.com*. In this case, if the user visits *365media.com* that makes a request to *aggregateintelligence.com*, this should not be cancelled.

I chose these lists because *EasyPrivacy* is the biggest and most known list created by community members. It should cover many web tracking threats. Ad blocking tools such as *uBlock Origin* relies on EasyList's *EasyList* to block ads. This tool has a big user base, and seems to be very successful.

I also chose Disconnect's list. This is because I believe it has a bigger potential to correctly categorising trackers. Both Disconnect and Mozilla are working on this list. It would also be interesting to compare them, basically comparing the community to these companies.

The blacklists are read by the extension, where a function reads each entry and sends them through an appropriate function for string-manipulation to add specific patterns that Chrome extensions can recognize. After that, the entries will be placed in a list. The tool observes all third-party HTTP requests and checks if a pattern in the list matches the URL in an HTTP request. If it does, and it's not whitelisted, the request will be cancelled and no information with the website can be exchanged. If it doesn't match, the request will go through as normal. We can observe this behaviour in Figure 4.3 and compare to the communication flow if the URL is not blocked in Figure 4.2. If *example.com* is in the list, the request is cancelled and *example.com* doesn't even receive the request. More importantly, *example.com* is not able to track the user at all.

Figure 4.3: Communication flow between a browser and a website when the browser extension cancels the request.

An important part of this solution is to not block first-party requests. Many legit domains such as *youtube.com* that many users visit directly is also used to track users in a third-party context. This domain is included in some lists. Therefore, I let all first-party requests go through, while only blocking third-party requests if they're in the blacklists.

With regards to the secondary goal specified in section 4.1, there's not much I can do to prevent breaking website functionality if a resource is incorrectly placed in either of the blacklists. However, the choice to only block third-party requests and not first-party requests, will let users directly visit potential tracking sites and prevent these from being unreachable.

4.2.2 Method 2. Third-Party Cookie Protection

By using the blacklist, many fingerprinters, web beacons and advanced cookie mechanisms are blocked. However, HTTP cookies are so widespread and practically present on every website. Blacklists can't include all of these, as that would prevent many requests to legit third-parties to be loaded. Hence, I also developed a method to prevent most third-party tracking cookies based on their prevalence. Websites can set cookies with the *Set-Cookie* HTTP header and read previously set cookies with the *Cookie* HTTP header. By observing HTTP messages, we can look into the headers.

This protection method removes cookie and referer headers from HTTP messages if the tool has seen the same domain attempting to set or read cookies in a third-party context on three different websites the user directly visits. I got the idea for this solution from Privacy Badger [99]. While Privacy Badger will block any tracking attempt (whether it is a script, image, fingerprinter etc.) if it has seen it on three different websites, this solution will only block cookies, while leaving the other tracking methods for the blacklist.

Lets assume *example.com* is a popular website and embeds trackers from *tracker.com* that use third-party cookies for this purpose. When a user visits *example.com*, this approach now observes *tracker.com* setting or reading cookies in a third-party context. The tool now knows *tracker.com* is a third-party tracker on one website, and stores information about this. If the user later visits another popular website, *example-second.com*, that also embeds trackers from *tracker.com*, this approach will again observe this and notice that *tracker.com* is a third-party tracker on two different websites. When this happens the third time, this approach will take action and deny this domain from setting or reading cookies ever again.

When a domain is registered as "cookie-blocked", it will be added to a list that is checked every time an HTTP message comes through. If it's in this list, it will loop through its headers and add every header that is *not* cookie or referer related into a new list. This new list is returned to the website, and does not contain the cookies. Referer headers are removed because they can be used to create a fingerprint, and to leak information (see section 2.3.2).

Another design decision that was made for this solution is to not "cookie-block" the whole domain if only a subdomain is observed using cookies. It is normal for websites to rely on third-parties for functionality such as authentication, user login, comment sections and much more. Assume *example.com* is a popular website that provides both important functionality and a tracking service to other websites. It has two subdomains, *func.example.com* to provide functionality and *track.example.com* to provide the tracking service. The tool will end up only removing cookies from *track.example.com*, and not from the whole domain (*example.com*). This, again, comes down to a trade-off: privacy vs. usability, and is directly tied to the secondary goal specified in section 4.1. Sometimes the same domains or subdomains are used for both services, in that case, this approach may end up breaking parts of the website. Unfortunately, I don't have a good solution to prevent this.

4.2.3 Do Not Track

The Do Not Track header (see chapter 3.1.3) is an optional header that can be sent to websites to let the user express if they want to be tracked or not. Some websites honor this initiative, but I don't know which. The header is therefore automatically sent to all websites. This is done by appending the new header *DNT=1* before sending the headers to a website.

4.3 User Interface

I created a user interface to let the user see how many trackers are actually blocked, and how many domains have been categorised as using tracking cookies and have been blocked from setting them. This interface is easily accessible by clicking the extension icon in the top right corner of the browser. In Figure 4.4 we can see how the popup window looks. I visited 30 popular websites to let the extension gain knowledge about domains that set third-party tracking cookies, and then visited *www.nytimes.com*. We can see that *nytimes.com* embed five different trackers that have been blocked. Clicking the button "Show all trackers" the user can see the full URL of the five blocked trackers. In case a website embed multiple trackers from the same domain, the simple list will just say "3 different tracker(s) from this domain:", but clicking the "Show all trackers" lets the user see all of them.

Below "Showing all cookie blocked domains:" the user can see the domains that the extension has blocked from setting and reading cookies. These domains will be blocked from setting third-party cookies on all websites the user directly visits.

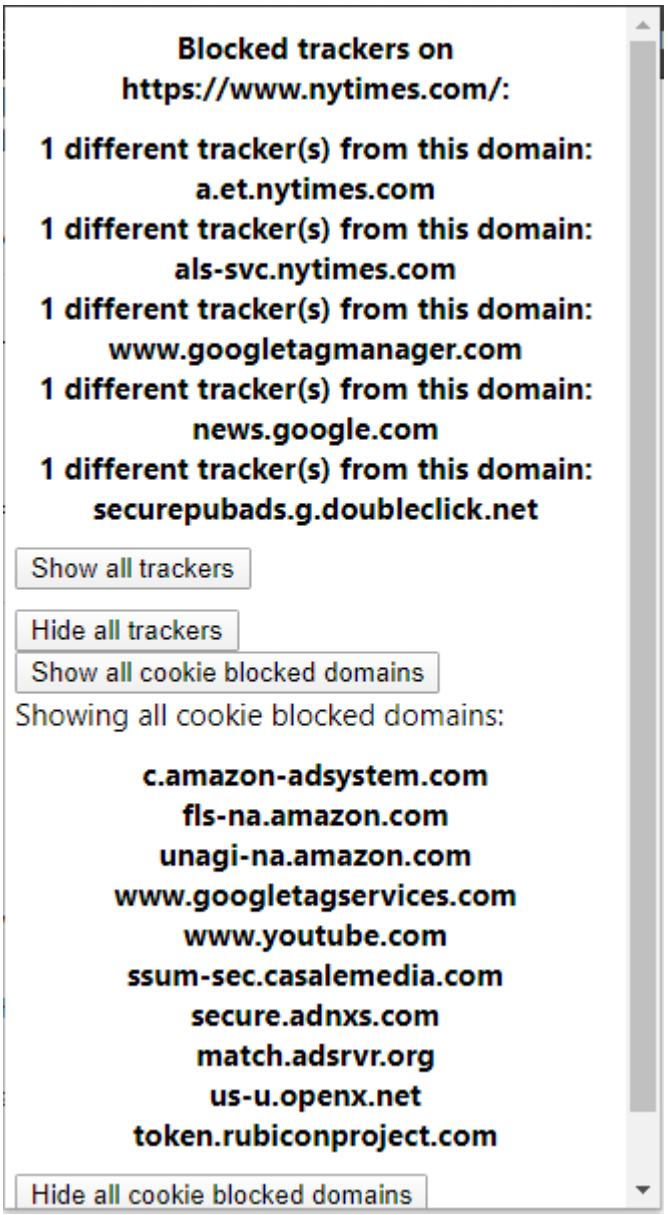


Figure 4.4: Popup (user interface) when visiting www.nytimes.com after having visited 30 popular websites prior, to gain knowledge about third-party tracking cookies.

4.4 APIs

chrome.webRequest

The extension mainly relies on the *webRequest* [28] API for observing and intercepting HTTP messages. If we look at Figure 4.5 we can see a set of events that follows the lifecycle of web requests. The *onBeforeRequest* is fired when a request is about to occur, before any TCP connections are made with the website [28]. This is the event that lets us cancel a request and is used if the requests match an entry in the blacklist.

The *onBeforeSendHeaders* event is also fired when a request is about to occur, but after the first headers have been created [28]. This is where the extension check for the *Cookie* and *referer* headers, as those will automatically be prepared if there is a cookie on the client's computer where the cookie-domain is equal to the request's domain. The Do Not Track header is also added to the rest of the headers here.

The event *onHeadersReceived* is the next important event, as this is fired every time the extension receives an HTTP response [28]. This is where the extension checks for *Set-Cookie* headers, and where applicable, remove these before they can reach the web browser.

In order for the extension to know which website the user currently visits (the first-party), this can also be known by utilizing the *webRequest API*. The first event that is fired, *onBeforeRequest*, also contains the type of the resource. If the request is type *main_frame*, it means that the request happens in the main frame, meaning the website the user directly visits. This is important to know as early as possible for two reasons: (i) so the extension doesn't block the request even if it is in the blacklist (this would block users directly visiting youtube.com for example), and (ii) for the correct first-party domain to be stored with the cookie information.

chrome.runtime

The *runtime* API [27] is used for various things, but in this extension I utilize the *sendMessage* and *onMessage* parts of the API. Because the popup script can't access variables and data within the background scripts, this data has to be sent with messages between them. When the user clicks the popup button, messages are sent to the background script asking for this data. The background script receives this message and sends a response containing the appropriate data. The popup script receives this data and shows it in the popup window.

chrome.extension

The *extension* API [26] is also used for various things, but in this extension only the *getURL* part is used. This lets the extension request local files (in this case the blacklists, which are .txt or .json files) and read them.

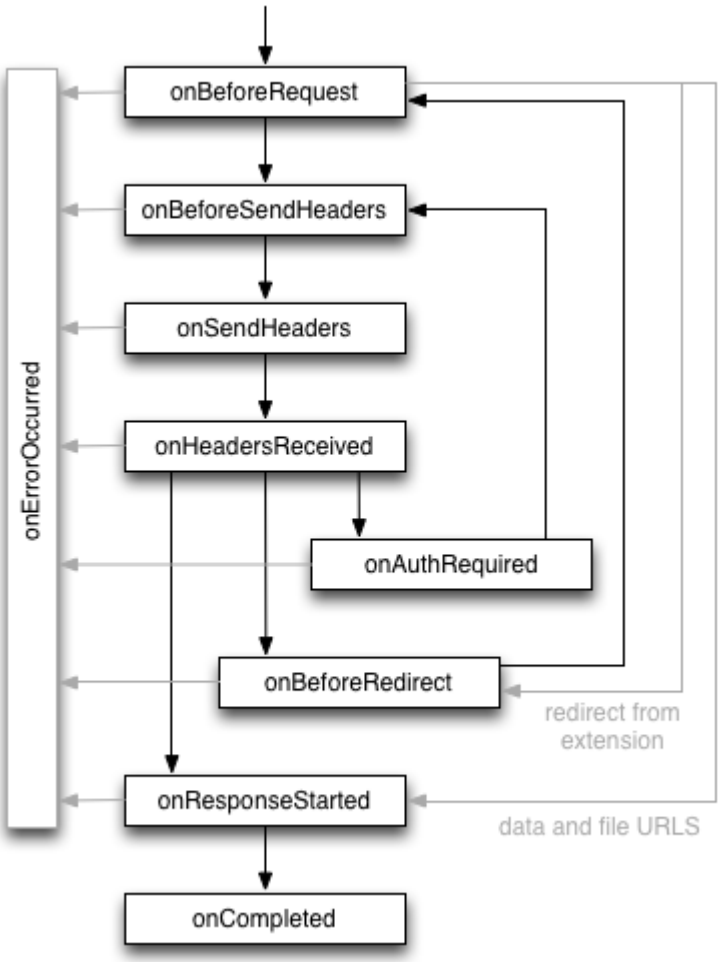


Figure 4.5: The lifecycle of requests. Figure taken from [28].

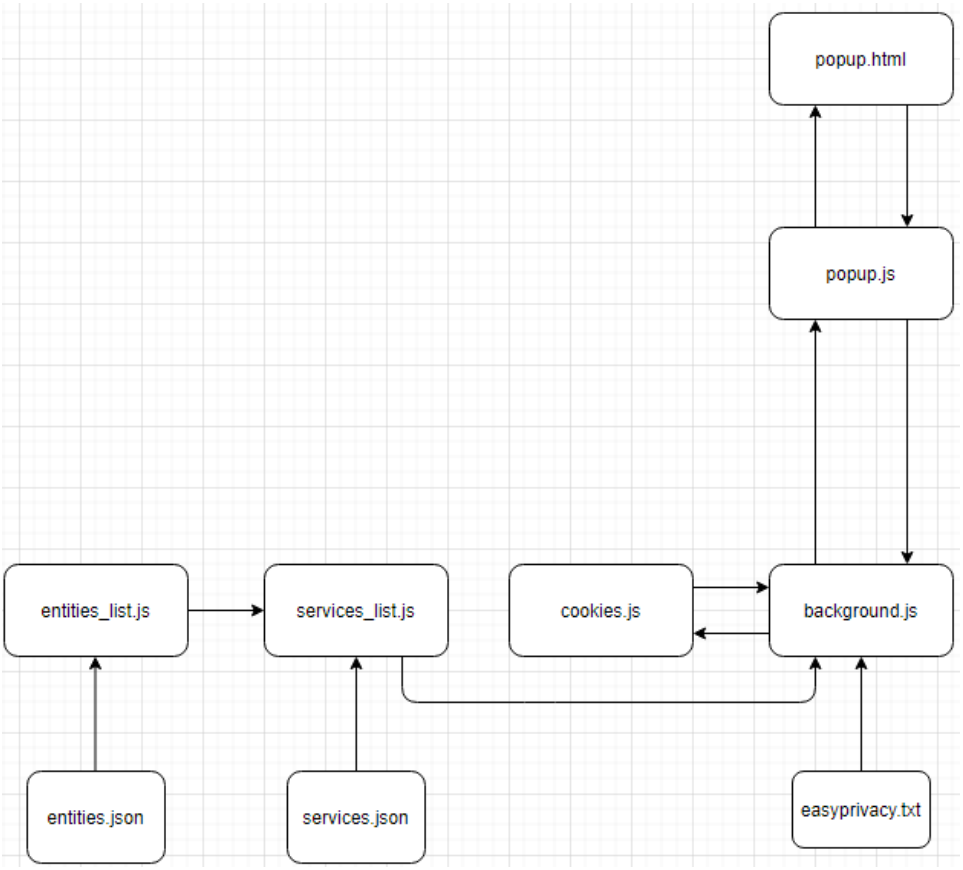


Figure 4.6: Overview of the internal structure of the extension, and the interaction between the components. The arrows indicates in which direction data flows.

4.5 Internal Components

This extension consists of a few scripts where each have their own responsibilities that is explained below. In Figure 4.6 we can also see an overview of the components and the interaction between them. The arrows indicate which direction data flows and between which components.

background.js

This is the main script that is loaded first. It is defined in the *manifest.json* file to be a background script. This script has three main responsibilities:

- The *easyprivacy.txt* list is loaded, each entry is sent through an appropriate pattern matching function, and then placed in a list. After this, a listener for the *onBeforeRequest* is created. The script is now observing all HTTP requests and determining if their domain is in the list.
- The script also observes requests looking for cookie-related headers. A part of this algorithm includes creating objects of type *CookieInfo*

that is within *cookies.js* and using its functions to determine whether to "cookie-block" a domain or not. To inspect headers, this script uses the *onBeforeSendHeader*s and *onHeadersReceived* events of the API.

- Sending messages using the *message passing* functionality that includes information about what trackers are blocked, and what domains are "cookie-blocked" to the popup.js script.

services_list.js

This is the second script to load, and is also defined in the *manifest.json* file to be a background script. This script has mainly one responsibility:

- The *services.json* file is loaded (which is one of the two blacklists provided by Disconnect), and the rest of the process is similar to the process in background.js. The difference is that Disconnect's lists are formatted differently than EasyPrivacy's list, so new functions are needed. Extra code is also needed to be able to whitelist some of the entries. A new *onBeforeRequest* listener and function is created. The reason I chose to use two separate ones, is because it would be easier to enable or disable one of the lists at a time, so comparing them in the next chapter would be easier.

entities_list.js

This is the next script to load, and is also defined in the *manifest.json* file to be a background script. This script has one responsibility:

- The *entities.json* file is loaded (which is the second blacklist provided by Disconnect), each entry sent through a pattern matching function, and then added in the same list as the entries in *services.json*. Extra code is also needed to be able to whitelist some of the entries. No new listener for *onBeforeRequest* is created, as it will use the same as in services_list.js. This is because I wanted to merge the Disconnect lists and treat them as one.

cookies.js

This is the next script to load, and is defined in the *manifest.json* file to be a background script. Although it doesn't need to be as no API calls are made here, it is easier, because background scripts can easily share data and use functions defined in another background script. This script is used to store information about cookies, domains and their prevalence. It contains a class *CookieInfo*, and background.js creates objects of this type and utilize its functions to store information, and then store these objects in a list. A *CookieInfo* object is created for every third-party domain that is observed attempting to set or read cookies. Its responsibilities are therefore:

- Storing information about third-party domains, first-party domains associated with a third-party domain, and the number of first-party

domains. It provides functions for background.js to check how many first-party domains are associated with a specific third-party domain, and to add new first-party domains to an existing third-party domain.

popup.html

This script is defined in the *manifest.json* file to be a popup file. It is loaded every time the user clicks the extension icon. Its responsibilities are:

- To add buttons and some text in the popup window.

popup.js

This script is not defined in the *manifest.json* file, but it provides functionality to popup.html. It is loaded every time the user clicks the extension icon. Its responsibilities are:

- Sending messages to the background script every time it is opened to ask for information about blocked trackers and "cookie-blocked" domains. These are stored in data structures inside the file. It contains functions to add listeners for the buttons seen in the popup window and to show the appropriate data when the button is clicked.

4.6 Implementation

This section goes through some of the key implementation details.

4.6.1 Blacklist-Based Approach

Extensions are very limited in terms of how much access to local files they have. In order to read in the blacklists, we have to make a GET request to the files, using XMLHttpRequest (XHR).

The extension uses three different blacklists and all of them are structured differently. In order to properly load them and be able to match URLs in HTTP requests, I had to add patterns to them that Chrome extensions recognize.

Pattern Matching

The *Match Patterns* [47] functionality is used in the blacklist solution. This lets us add specific patterns before, in the middle of, or after an entry in the blacklist to match a URL.

- We can add ***://** to match both http and https.
- We can add * that acts as a wildcard. This can be used at the start of the hostname to match any subdomain for a given domain, it can replace the hostname altogether to match any domain, or after the domain to match any path. It can also be used anywhere in the path.

A problem with the EasyPrivacy list and my implementation was that a few entries (around 25 out of the 16,000) is specified with a * in the middle of the hostname, e.g. *www.example.*.second-example.com*. According to Chrome and *match patterns*, this is not allowed. I later found out that other tools that utilize these types of lists rely on regular expressions to match patterns, in order to circumvent this restriction. I noticed this too late, and decided to stick with my original implementation. This means that those 25 entries in this list is not working.

A couple of simple examples from the EasyPrivacy blacklist can be:

mixpanel_tracker.
abcstats.com

In order to properly pattern match these entries, we can alter and change them to this:

://*/*mixpanel_tracker.

This will now match any domain that uses http and https, and includes mixpanel_tracker. anywhere in the path. For the next entry, we can do this:

://abcstats.com/

This will now match all paths in the abcstats domain.

The entries in EasyPrivacy are categorised by what patterns should be added. The entries have a specific character as first and/or second character to specify this. They can start with:

- &
- -
- .
- /
- ;
- =
- ?
- _
- ||

This means that when the list is loaded and read, every first and/or second character is checked. If they match one of these, the entry will be sent to an appropriate function to add correct patterns and remove the first

and/or second character, among other things.

When this process is done and the finished entries are added in the list, a listener has to be set that listens for HTTP messages, like here:

```
function setListener_easyprivacy(){
    chrome.webRequest.onBeforeRequest.addListener(
        blockTrackers,
        {urls: easyprivacyList},
        ["blocking"]
    );
}
```

easyprivacyList is a list that contains all the entries with correct pattern matching and *"blocking"* means that the tool can intercept HTTP messages and cancel or redirect them. *blockTrackers* is the function where the cancelling happens, and is defined like this:

```
function blockTrackers(details){
    let type = details.type;
    let url = details.url;
    let url_hostname = getHostnameFromRegex(url);
    let currUrl_hostname = getHostnameFromRegex(currUrl);

    if(type == "main_frame" || url_hostname.includes
      (currUrl_hostname)){
        return {cancel: false};
    } else{
        blockedUrls.push(url_hostname);
        blockedUrlsFull.push(url);
        return {cancel: true};
    }
}
```

The function parameter *details* contains much information about the request. For example, we can get the URL by calling *details.url*. We can also see the request type. If it is *main_frame*, it means it is the URL the user visits directly and shall not be cancelled. The same applies if the request goes out to a subdomain of the current URL. For instance: if the user visits *example.com* that makes a request to *func.example.com* that provides some kind of functionality, this is technically a third-party request. Therefore I also check that the request doesn't include the current URL's domain. Otherwise, the function returns *cancel: true*, which means the request is to be cancelled before leaving the browser.

In order to get the current URL (the URL the user is currently visiting), a similar function is created earlier in the code. It checks whether the request is *main_frame*, and sets the variable *currUrl* to be *details.url* in that case. This way the current URL is updated every time the user visits a new page.

blockedUrls and *blockedUrlsFull* contains information about the trackers that are blocked on the current website, there is also a check here to check

whether the tracker is already in the lists or not. The lists are then sent to popup.js (this process is further explained in the section *Sending Information to popup.js*).

Disconnect's lists are a bit different. The entries are read and pattern matched similarly to the function for *EasyPrivacy*. Every entry is added to match the whole domain, any subdomain, and any path, and added to a list. A listener similar to the function for *EasyPrivacy* is also set, to listen for HTTP requests. However, the whitelisting part adds some more complexity. During the process of reading the *services.json* file and pattern matching the entries, the function also looks for domains that have other domains below it, specified as their "whitelisted domains". These domains are added to a map as the key. The function then looks for the domains that are supposed to be whitelisted for that particular domain, and adds these to a list. When all the correct domains have been added to this list, it's added to the map as a value. For instance, using the example provided in section 4.2.1, *twitter.com* and the five domains specified below it is added to the list of trackers with correct patterns. But *twitter.com* is also added to the map as a key. It's corresponding value will be a list containing the five domains specified below it.

The process for the *entities.json* file is executed after the tool has finished handling the *services.json* file. Its process is similar, but also have some differences: when the function finds an organization, it checks the entries specified as part of this organization to see if the organization's name is in one of the domains. If it is, it checks whether this domain already exist in the map as a key. If it does, the rest of the domains that are part of this organization is added to the value list of that key, because they're supposed to be whitelisted for that domain. If the domain doesn't exist as a key in the map, it gets added to the map, similarly to how its done for the *services.json* file. Last, if an organization's name is not included in any of the domains specified as part of this organization, I can't know the organization's main domain, and therefore this part won't be implemented. Luckily, though, this only the case for a small portion of the list.

The function that either cancels a request or not, is defined like this:

```
function blockServicesAndEntities(details){
    let details_url = details.url;
    let details_type = details.type;
    let url_hostname = getHostnameFromRegex(details_url);
    let currUrl_hostname = getHostnameFromRegex(currUrl);

    if(details_type == "main_frame" ||
       url_hostname.includes(currUrl_hostname)){
        return {cancel: false};
    } else{
        let cancelRequest = checkWhitelisting(details_url);

        if(cancelRequest){
```

```
            blockedUrls.push(url_hostname);
            blockedUrlsFull.push(details_url);
        }
        return {cancel: cancelRequest};
    }
}
```

The first part of this function is similar to that of *easyprivacy*. If the request is to a website the user directly visits, or the request is a subdomain of it, it won't be cancelled.

The variable *cancelRequest* contains a boolean value, *true* or *false*, that is returned by the function *checkWhitelisting()*. *details_url* is the current request URL to be checked. If the return value is true, it means the request shall be cancelled, and the blocked URLs are added to the same lists as in the function for *easyprivacy*. If the value is false, it means that the request is whitelisted and won't be blocked. The function *checkWhitelisting* is defined like this:

```
function checkWhitelisting(url){
    let list = [];

    for(let key in map){
        if(currUrl.includes(key)){
            list = map[key];
            break;
        }
    }

    for(let i = 0; i < list.length; i++){
        let element = list[i];
        if(url.includes(element)){
            return false;
        }
    }
    return true;
}
```

This function first checks if the current URL is specified in Disconnect's lists with whitelisted domains below it, by checking if the current URL exists in the map as a key. If it does, *list* is set to be the key's value in the map, the list of whitelisted domains.

The second loop checks whether or not the requested URL includes one of the whitelisted domains. If it does, the function returns with false, indicating that the requested URL shall not be blocked. Otherwise, the function returns true, and the request will be cancelled.

Sending Information to popup.js

background.js and popup.js can't communicate directly, and popup needs to know about the current URL (the URL the user is currently on) and

what trackers are blocked on this URL, in addition, an option to show the full URL of a tracker is also given to the user. As previously stated, the current URL is already known in the variable *currUrl*. In the code examples above, we can see that the blocked URL's hostname is pushed to a list *blockedUrls*. The complete URLs are added to another list, *blockedUrlsFull*. The information of blocked trackers in these lists only applies to one website. These lists are cleared in the same function that sets the current URL, in order to only show blocked trackers on the current website.
When this information is ready it can be sent to popup.js using *message passing*. Because the popup script is only loaded whenever the user clicks the extension icon, popup.js has to send a message to background.js and ask for this information when that happens. This can be done in the following way:

```
chrome.runtime.sendMessage({greeting: "currurl"},
function(response){
    currUrl = response.farewell;
    document.getElementById("currurl").innerHTML = currUrl;
});
```

greeting is the name of the message that is sent. It can be seen as a variable holding a value. The value is the string "currurl", to indicate to background.js that popup.js wants to know the current URL. The variable *currUrl* is set to *response.farewell*, which is the value of the variable *farewell* that is received as the response to this message. The last part of the code interacts with the popup.html to show this information in the popup window.

The background.js script listens for message events, and responds like this:

```
chrome.runtime.onMessage.addListener(
    function(request, sender, sendResponse){
        if(request.greeting == "currurl"){
            sendResponse({farewell: currUrl});
        }
    }
);
```

In this case, the *currurl* message is received to let background.js know this is the information popup.js wants. It sends a response containing the *currUrl* variable, that holds the current URL.

The same logic applies when sending the lists of blocked trackers as well, by using multiple one-time messages.

4.6.2 Third-Party Cookie Protection

For this method, I need two listeners, one to listen for incoming HTTP headers and one for checking outgoing HTTP headers before they go out. The listener checking outgoing headers is set like this:

```
chrome.webRequest.onBeforeSendHeaders.addListener(
    onBeforeSendHeaders, { urls: ["http://*/*", "https://*/*"] },
     ['requestHeaders', 'extraHeaders', 'blocking']
);
```

The other listener is similar, but using the *onHeadersReceived* event instead. These listeners also uses the pattern matching functionality to make them work on every website. I need to specify *requestHeaders* or *responseHeaders* to get access to the headers, *extraHeaders* is needed in the newer versions of Chrome (from version 72 and onwards) to get access to cookie and referer headers. *blocking* is needed to, again, be able to modify the headers. The listener specifies the function *onBeforeSendHeaders* which is where the cookie and referer removal happens.

The *onBeforeSendHeaders* function is defined as follows:

```
function onBeforeSendHeaders(details){
    let url = details.url;
    let url_hostname = getHostnameFromRegex(url);
    let currUrl_hostname = getHostnameFromRegex(currUrl);

    if(!url_hostname.includes(currUrl_hostname)){
        for(let i = 0; i < details.requestHeaders.length; i++){
            if(details.requestHeaders[i].name.toLowerCase()
                === "cookie"){
                    addUrls(url_hostname);
                    break;
            }
        }
    }

    isInCookieBlocklist(url_hostname, function() {
        // check if the domain is in the cookie-blocked list,
        // with a callback
    });

    if(isInList){
        let newHeaders = [];
        for(let i = 0; i < details.requestHeaders.length; j++){
            if(details.requestHeaders[i].name.toLowerCase()
                !== "cookie" &&
                details.requestHeaders[i].name.toLowerCase()
                !== "referer"){
                    newHeaders.push(details.requestHeaders[i]);
            }
        }
        return {
            requestHeaders: newHeaders
```

```
        }
    } else{
        return {
            requestHeaders: details.requestHeaders
        };
    }
}
```

As discussed in the design section, if a subdomain sets cookies, I only block the subdomain, and not the whole domain from setting or reading cookies. Therefore I grab the hostname (which will also include the subdomain) from a regular expression function. I only focus on third-party cookies, so I need to make two checks before registering the domain as using tracking cookies: make sure that the domain is not the first-party, and secondly, make sure that it actually sets or reads cookies. If both of these are true, the function *addUrls*, which starts the algorithm to register the domain as using tracking cookies, is called with that domain as input.

The function *isInCookieBlocklist* is called to check if the request domain is "cookie-blocked", and sets a boolean variable *isInList* to be true in that case. If the value is true, the function will create a list called *newHeaders* and loop through all the headers. Headers have a name and a value. If the header name is not *cookie* or *referer*, the header will be pushed to this new list. This way the cookie and referer headers are removed. The *newHeaders* list is then returned and sent to the website. If *isInList* is false, the function will just return the headers normally.

The function *onHeadersReceived* that deals with incoming headers, works the same way. The only difference is that instead of looking for and removing cookie and referer headers, the function removes *set-cookie* header. This is the header that is used to set cookies on the client's computer.

As mentioned, the *addUrls* function is used to register domains as using tracking cookies, and determines if a domain (or subdomain) shall be "cookie-blocked" or not. *cookies.js* contains a class *CookieInfo* that is created for every third-party domain. It's initialized with a constructor that takes input a third-party domain and a first-party domain (current URL), and initializes a list of first-party domains with this appended to it. The class has three functions:

- *getTpdomain()* that returns the third-party domain associated with this object.
- *checkIfDomainExists(firstpartyDomain)* which returns true if the input domain is in this local first-party list.
- *addDomain(firstpartyDomain)* that uses the above function to check if it has seen this domain before. Otherwise it's appended to the list. If the list of first-party domains now contains 3 elements, this function returns 1. Otherwise it returns 0.

The *addUrls* function in the background.js script is defined like this:

```
function addUrls(url){
    if(cookieInfoList.length == 0){
        cookieInfoList.push(new CookieInfo(url, currUrl));
    } else{
        let i = checkIfTpExists(url);
        if(i >= 0){
            let u = cookieInfoList[i].addDomain(currUrl);
            if(u == 1){
                let tpdomain = cookieInfoList[i].getTpdomain();
                if(!checkIfCookieBlocked(tpdomain)){
                    cookieBlockedDomains.push(tpomain);
                }
            }
        } else{
            cookieInfoList.push(new CookieInfo(url, currUrl));
        }
    }
}
```

The *cookieInfoList* contains objects of *CookieInfo* and the *cookieBlockedDomains* list contains the "cookie-blocked" domains in a string format.

The function takes input a third-party domain (url), and if this is the first time the extension is run, the list of objects is empty. If it is, an object is created with the third-party's domain and the current URL's domain. Otherwise, a check is made to see if this third-party domain has already been observed. The *checkIfTpExists* function loops through the *cookieInfoList* and returns the location in the list where this object is, as an integer (i). If it doesn't exist, i will be -1 and the if-check is false. An object is thereby created for this third-party domain. If it exists, the tool now knows the location in the list stored in variable i. It then attempts to add the current URL to this third-party by calling its function *addDomain*. If *addDomain* returns 1, it means this third-party is seen on three different first-parties and should be blocked. Otherwise, it returns 0. So the tool checks if this variable (u) is 1. If it is, it grabs its third-party domain in a string format and adds it to the list of cookie blocked domains if it's not already in the list.

In order for the user interface to show users which domains that are registered as "cookie-blocked", it asks *background.js* for this information as well when the popup button is clicked, and is then sent to the popup script. This works using the *message passing* functionality, similar to how it was explained in section 4.6.1.

4.6.3 Do Not Track Header

This header is pushed to the list containing all other headers in the function *onBeforeSendHeaders*, like this:

```
details.requestHeaders.push({name: "DNT", value: "1"});
```

If a domain is "cookie-blocked", this header is pushed to the *newHeaders* list instead.

Chapter 5

Evaluation

This chapter investigates how well my tool and the state of the art works.

5.1 Testing on Panopticlick

The first part of the evaluation is testing the tools on a website called *Panopticlick* [97]. This website is a research project and created by the Electronic Frontier Foundation (EFF). The goals of the project are to discover tracking techniques and test privacy protecting tools [1]. The website creates environments to simulate how real trackers work, and observes if the browser manages to stop the tracking attempts. Multiple tracking domains are created as third-party trackers, and requests to these domains are made. This website can be visited with any of the tools enabled. When the test button is clicked, the website runs five tests:

- *"Is your browser blocking tracking ads?*
- *Is your browser blocking invisible trackers?*
- *Does your blocker stop trackers that are included in the so-called "acceptable ads" whitelist?*
- *Does your browser unblock 3rd parties that promise to honor Do Not Track?*
- *Does your browser protect from fingerprinting?"* [97]

The first test creates a visible ad that is used for tracking purposes, and tries to set a cookie. If the ad is blocked, the test is passed [1].

The second test creates a web beacon that is not visible. If the request to this resource is blocked, the test is passed [1].

In the fifth test the uniqueness of the browser is tested. Panopticlick uses different fingerprint techniques to gather a total of 14 data points about the browser and client's machine. Based on this information, it checks whether the user is unique compared to other Internet users' configurations [1]. If the uniqueness is above a certain threshold, the test is passed.

I enabled each tool and tested them on the website one by one. The

Test/Tool	My tool	Ghostery	Disconnect	Privacy Badger	Firefox Content Blocking	Brave
Is your browser blocking tracking ads?	Yes	Yes	Yes	Yes	Yes	Yes
Is your browser blocking invisible trackers?	Yes	No	Yes	Yes	Yes	Yes
Does your blocker stop trackers that are included in the so-called "acceptable ads whitelist"?	Yes	Yes	Yes	Yes	Yes	Yes
Does your browser unblock 3rd parties that promise to honor Do Not Track?	No	No	No	Yes	No	No
Does your browser protect from fingerprinting?	No, browser has a unique fingerprint	No, browser has a unique fingerprint	No, browser has a unique fingerprint	No, browser has a unique fingerprint	No, browser has a unique fingerprint	No, browser has a unique fingerprint

Figure 5.1: Result of the tests on Panopticlick for each tool.

results can be seen in Figure 5.1. It seems that blocking the tracking attempts in test 1 and 2 is easy for all tools, except for Ghostery. Strangely enough, it scored *no* on the second test. It's hard to understand why seeing as Ghostery scores very high on studies conducted in previous research. I can see two potential reasons why this is: (i) the simulated tracking isn't behaving close enough to how real trackers behave, or (ii) Ghostery's blacklist is too lenient, and their algorithmic approach (enhanced anti-tracking) doesn't recognize this as a tracking attempt (this could be related to (i)).

Apart from this, every tool blocks ads even if they are in the *acceptable ads whitelist*. Ads that are not considered intrusive or annoying, does not perform third-party tracking and abide by a certain standard, are placed in this whitelist [6].

Privacy Badger is the only tool that checks whether a domain honors the *Do Not Track* policy, and disables anti-tracking on these domains.

When it comes to fingerprinting, this result shows unfortunate, but not surprising results. None of the tools are able to create a non-unique browser fingerprint. This demonstrates the difficulty in this problem.

5.2 Testing Against Common Trackers

A part of the collaboration work between Guleed Abdi and me was for him to set up a website with some web tracking techniques implemented, and

for me to test how well my tool perform on this website. Abdi set up a website called *www.guleedsthesis.online* with three trackers implemented:

- Facebook Pixel
- Google Analytics
- IPinfo

Facebook Pixel is Facebook's analytics tool. It can be implemented on websites to collect data from users to optimize ads, more effectively be able to hit targeted audience for future ads and remarket to people who already visited the website [117]. The tracking techniques used by Facebook Pixel is first- and third-party cookies, as well as a form of cookie sharing technique. This is implemented as a tiny image, and adds cookie information to its URL parameters.

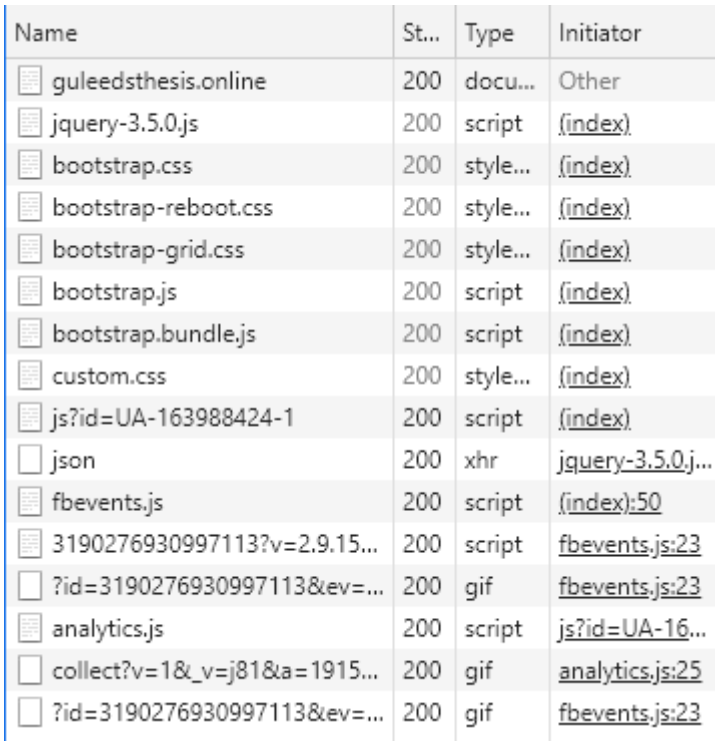
Google Analytics is the most common analytics engine used to analyse traffic. How Google Analytics works was explained in section 2.4 (*Tracking Types*). Google Analytics is often used together with Google Tag Manager. Google Tag Manager allows developers to add marketing tags (code snippets or tracking pixels) to a website. The data gathered by Google Tag Manager is shared with Google Analytics. This process uses the same idea as Facebook Pixel, by leaking cookie information back to *google-analytics.com* by making requests to a tiny image, and including cookie information in its URL parameter [67].

IPinfo (ipinfo.io [80]) is a service that can be used to collect data from users' IP address. This information includes hostname, city, region, country, geolocation (coordinates), postal code, timezone of the user. It can also find information about a user's internet service provider [80]. IPinfo maintains an API that can be used with several programming languages.

I visited *www.guleedsthesis.online* with no defence systems turned on. The IPinfo method was able to gather all the information mentioned above.

If we look at Figure 5.2 we can see that the website makes HTTP requests to multiple scripts. Four of them are associated with Facebook (one with name fbevents.js and three with initiator fbevents.js) and two requests to *google-analytics.com* (one with name analytics.js and one with initiator analytics.js) as well as one request to *googletagmanager.com* (the one with name js?id=UA-163988424-1). There is also one request to *ipinfo.io* (name json).

If we look at Figure 5.3 we can see all the cookies that are set with their associated domain. Facebook sets a lot of cookies in a third-party manner, as well as one first-party cookie with name *_fbp* and domain *guleedsthesis.online*. Google Analytics can be implemented without the usage of third-party cookies, which is done in this case. This means it only sets first-party cookies, that we can see in Figure 5.3. The three cookies with name *_gid*, *_ga* and *__utma* are Google Analytics cookies. If we look further into the HTTP requests and cookies, we can observe how the first-party cookies are leaked back to either Facebook or Google Analytics. Lets again look at



Name	St...	Type	Initiator
guleedsthesis.online	200	docu...	Other
jquery-3.5.0.js	200	script	(index)
bootstrap.css	200	style...	(index)
bootstrap-reboot.css	200	style...	(index)
bootstrap-grid.css	200	style...	(index)
bootstrap.js	200	script	(index)
bootstrap.bundle.js	200	script	(index)
custom.css	200	style...	(index)
js?id=UA-163988424-1	200	script	(index)
json	200	xhr	jquery-3.5.0.j...
fbevents.js	200	script	(index):50
3190276930997113?v=2.9.15...	200	script	fbevents.js:23
?id=3190276930997113&ev=...	200	gif	fbevents.js:23
analytics.js	200	script	js?id=UA-16...
collect?v=1&_v=j81&a=1915...	200	gif	analytics.js:25
?id=3190276930997113&ev=...	200	gif	fbevents.js:23

Figure 5.2: Visiting guleedsthesis.online with no defence systems on. Screenshot of the HTTP requests made by the website.

Figure 5.2. We can see three requests to a gif-type resource. Two of these are associated with Facebook, while one is from Google Analytics. These gifs are the tracking pixel, implemented as an image and the way they leak the first-party cookies back to Facebook or Google Analytics. The full URLs which sends these cookie-values back to their home is too long, but here is a shortened example:

https://www.google-analytics.com/r/collect?v=&_gid=1140343996.1588167820

In this example we can see that the value for the cookie *_gid* in Figure 5.3 is the same as the value in the URL parameter for the request to *google-analytics.com*.

I then ran three tests on his website with different functionality of my tool enabled or disabled.

Test 1

For this test I turned off the blacklist functionality and only kept the third-party cookie protection method on, but without training it beforehand. Without training the tool first, it doesn't know about any domains that use tracking cookies. The results were therefore the same as without using any countermeasures, which was no surprise.

Name	Value	Domain
fr	1RgnLwrZXrdBHkRRA.AWVuVXW4eM0FNXiXaaicOTxnmr4.Bele4E....	.facebook.com
spin	r.1002052802_b.trunk_t.1588110977_s.1_v.2_	.facebook.com
wd	1920x937	.facebook.com
xs	41%3A4hyYiz8lR9on8Q%3A2%3A1586884123%3A6308%3A5914	.facebook.com
_fbp	fb.1.1587982694347.133952972	.facebook.com
sb	CO6VXv8XPAcuvWQAkWIKvuZo	.facebook.com
datr	-e2VXo7_LWD6ys3a0ehmQhzS	.facebook.com
c_user	706985976	.facebook.com
__utma	44433727.479997343.1531309788.1531309788.1531309788.1	.jquery.com
_gid	GA1.2.1140343996.1588167820	.guleedsthesis.onl...
_ga	GA1.2.601450071.1588167820	.guleedsthesis.onl...
_fbp	fb.1.1588167819843.851713135	.guleedsthesis.onl...

Figure 5.3: Visiting guleedsthesis.online with no defence systems on. Screenshot of the cookie storage found in developer console.

Test 2

For this test I again turned off the blacklist functionality and only kept the third-party cookie protection method on. However, this time I trained it beforehand on the top 100 websites in the Tranco list [120]. This let the tool observe many third-parties attempting to set or read cookies and register many domains as using tracking cookies.

The results were that IPinfo could still gather data about my computer and network. This is because IPinfo doesn't use cookies.

When it comes to Google Analytics and Facebook Pixel, the same HTTP requests that we observed in Figure 5.2 still took place. However, Facebook Pixel and Google Analytics are very common analytics engines and are utilized by many websites, so these were registered by the tool as tracking attempts. We can see the result in Figure 5.4. All third-party cookies were removed, which thereby removes their ability to track us cross-site. Unfortunately, though, my cookie protection method does not focus on first-party cookies, so we can see that Google Analytics were able to set three cookies (*__utma*, *_gid* and *_ga*) and Facebook one cookie (*_fbp*).

Name	Value	Domain
__utma	44433727.479997343.15313097...	.jquery.com
_gid	GA1.2.1952255739.1588186612	.guleedsthesis.online
_ga	GA1.2.1416403985.1588186612	.guleedsthesis.online
_fbp	fb.1.1588186611521.1517897232	.guleedsthesis.online

Figure 5.4: Visiting guleedsthesis.online with trained cookie-protection on. Screenshot of the cookie storage found in developer console.

Test 3

For the last test I turned on the blacklist functionality, and thereby using the tool to its full potential. If we take a look at Figure 5.5 we can see that three HTTP requests were cancelled/blocked before they could go out. These requests were to *googletagmanager.com*, *connect.facebook.net* and *ipinfo.io*. Because these were blocked, no information could be collected by IPinfo. Further, no requests to Google Analytics could be made (it relies on googletagmanager.com to respond first) and no further request went to Facebook. In Figure 5.6 we can now see that the cookie storage is empty and no cookies could be set by either of the tracking services.

Name	Status	Type	Initiator
guleedsthesis.online	200	document	Other
custom.css	200	stylesheet	(index)
bootstrap.bundle.js	200	script	(index)
bootstrap.js	200	script	(index)
bootstrap-grid.css	200	stylesheet	(index)
bootstrap-reboot.css	200	stylesheet	(index)
bootstrap.css	200	stylesheet	(index)
jquery-3.5.0.js	200	script	(index)
fbevents.js	200	script	(index):50
js?id=UA-163988424-1	(blocked:other)	script	(index)
3190276930997113?v=2.9.18...	(blocked:other)	script	fbevents.js:23
json	(blocked:other)	xhr	jquery-3.5.0.js:10099

Figure 5.5: Visiting guleedsthesis.online with all protection mechanisms enabled. Screenshot of the HTTP requests made by the website.

Name	Value	Domain

Figure 5.6: Visiting guleedsthesis.online with all protection mechanisms enabled. Screenshot of the cookie storage found in developer console.

Conclusion

The conclusion based on these tests is that my tool provides full protection and protects users' privacy when users visit websites that utilize analytics services from Google Analytics and Facebook Pixel, as well as those websites that utilize services from the IPinfo API.

5.3 Evaluation and Comparison with State of the Art

This section presents an experiment that involves running the tool on multiple websites. Because we can't possibly know how many trackers are connected to a website, the best metric to use is to compare it with similar tools.

Setup

This experiment consists of running my tool, Ghostery, Disconnect, Privacy Badger, Firefox and Brave on the same websites. My tool, Ghostery, Disconnect and Privacy Badger were tested on Google Chrome. Unfortunately I couldn't find a way to automate these tests, where the amount of blocked trackers for each of the tools were saved automatically. This is therefore a manual experiment. For this reason, I picked 30 websites out of the top 70, that were chosen from the Tranco list of top 1 million websites. There are three reason for not picking exactly the top 30 websites:

- Some websites didn't respond.
- None of the protection tools discovered any trackers on some of the websites.
- Some of the websites are whitelisted by default by some of the tools, and therefore didn't block any trackers. I thought it would be more fair to choose websites where every tool blocked at least one tracker.

The 30 domains can be observed in Table 1.

mozilla.org	flickr.com	ebay.com	github.com
goo.gl	msn.com	dropbox.com	okezone.com
amazon.com	linkedin.com	apple.com	bit.ly
yahoo.com	fandom.com	adobe.com	doubleclick.net
taobao.com	vk.com	imgur.com	quizlet.com
vimeo.com	pinterest.com	giphy.com	jd.com
weibo.com	reddit.com	wordpress.com	yelp.com
google-analytics.com		amazonaws.com	

Table 5.1: The 30 visited domains.

The experiment is done by disabling all other extensions, and just enabling the one to currently test. Browsing history and browser cache is deleted between testing each tool.

Configuration of Each Tool

Most of the tools have multiple settings and configuration options to determine the strictness or aggressiveness of the anti-tracking mechanisms, so it's important to check these settings. Below are the settings chosen for each tool:

- **Ghostery** is tested with all countermeasures enabled.
- **Disconnect** exists as a free and premium version (which comes with more features), but I have tested the free version and all its countermeasures are enabled.
- **Firefox Content Blocking** is set to custom with all anti-tracking mechanisms enabled.
- **Privacy Badger** has all countermeasures enabled. Because Privacy Badger isn't as good out of the box and needs to be trained to perform on par with the others, I first trained it on the top 100 websites.
- **Brave** is tested with the following settings enabled:
  - Block cross-site trackers.
  - Block cross-site fingerprinting.
  - Disallow social media tracking.
  - Block cross-site cookies. The only available options here is to either block all cookies, accept all cookies, or only block third-party cookies, so I chose the latter.

My tool relies on a blacklist, and an algorithmic approach to cookies. Because the cookie-protection approach doesn't know of any tracking cookies by default, it needs to be trained. This is done similarly to how I trained Privacy Badger, by first visiting the top 100 websites in the Tranco list, and then do the testing on the 30 websites. If we look back at Figure 4.5, we see that the event where the cancelling of requests happen, is before the event that deals with HTTP headers, such as cookie headers. Because of this, if a tracker that rely on HTTP cookies is specified in a list, the request will be cancelled before the tool can detect the use of cookies. I therefore did two tests with my tool:

- Test 1 is conducted with both blacklists enabled, as well as after having trained the tool on the top 100 websites.
- Test 2 is conducted by disabling both blacklists, to get a better view of how the cookie-protection method performs. The tool is trained on the top 100 websites prior to this test also.

Definition of *one* Tracker

I noticed that on some websites, my tool cancels requests to tens or hundreds of URLs, while the other tools doesn't. For this reason, I looked more into how I should measure the number of trackers that it and the other tools block. I made three decisions regarding this:

1. If two or more blocked trackers' URL is 100% equal, I treat it as one tracker.

2. If two or more blocked trackers' domain is equal, but the path of the URL is different, I treat them as different trackers.

3. If two or more blocked trackers' URL is equal, but their URL parameters are different, I treat them as different trackers. This is because even though the domain and most of the path is the same, one can be used to leak a cookie value in its URL parameter, while the other can be used to leak other sensitive information (such as the user's e-mail address).

Results Test 1

The results of the first test can be seen in Figure 5.7. As we can see, most of the tools block roughly the same amount of trackers, with Disconnect being a little low, my tool a little higher, and Brave with a great result.

Disconnect and Firefox are utilizing the same blacklists, so one could assume they should block about an equal amount of trackers. However, Firefox also intercept cookies from non-blacklisted websites. This could potentially be the reason they stop a little more trackers.

Brave is stopping a lot of trackers, but they do have more anti-tracking mechanisms than other tools. They also block all third-party cookies in my test, so their blocking might be too aggressive. However, none of the state of the art tools visibly broke any of the tested websites.

When it comes to my tool, it does show promising results. The tool observed a total of 5 domains or subdomains that were registered as using tracking cookies after it had been trained on the top 100 websites. During the testing, 176 of the stopped tracking attempts came from cancelled requests by the blacklists. The remaining 7 tracking attempts were via cookies, and these were stripped away by the cookie-protection method.

Results Test 2

The results after test 2 can be observed in Figure 5.8. After having trained it on the top 100 websites, 627 unique domains or subdomains were observed attempting to set or read cookies. In addition, 60 of these were registered as "cookie-blocked", because they were observed using cookies in a third-party context on at least 3 different websites.

During the testing on the 30 websites, 166 unique-per-website domains or subdomains were stopped from setting or reading cookies. These domains attempted to read or set cookies multiple times, though, reaching about 1,000 tracking attempts in total, which all were stopped.

I couldn't see any visible breakage to websites by having these cookies removed, but it could be the case if I tested some of the functionality, such as log-ins. However, the vast amount of cookies that were removed came from known tracking and analytics domains, such as *google-analytics.com*, *googletagmanager.com*, *amazon-adsystem.com*, *facebook.com*, *scorecardresearch.com* and *ib.adnxs.com*.

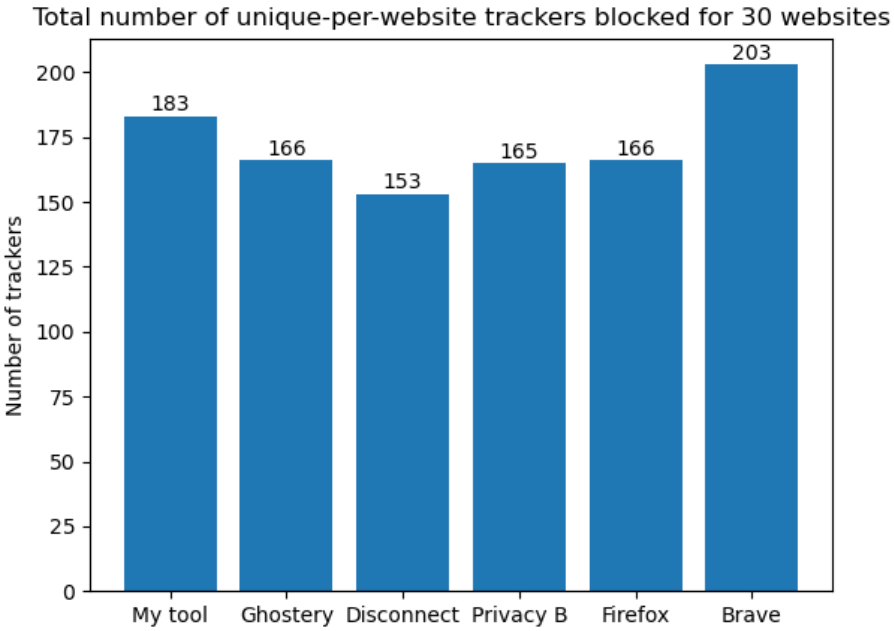


Figure 5.7: Total number of unique-per-website trackers blocked for 30 websites.

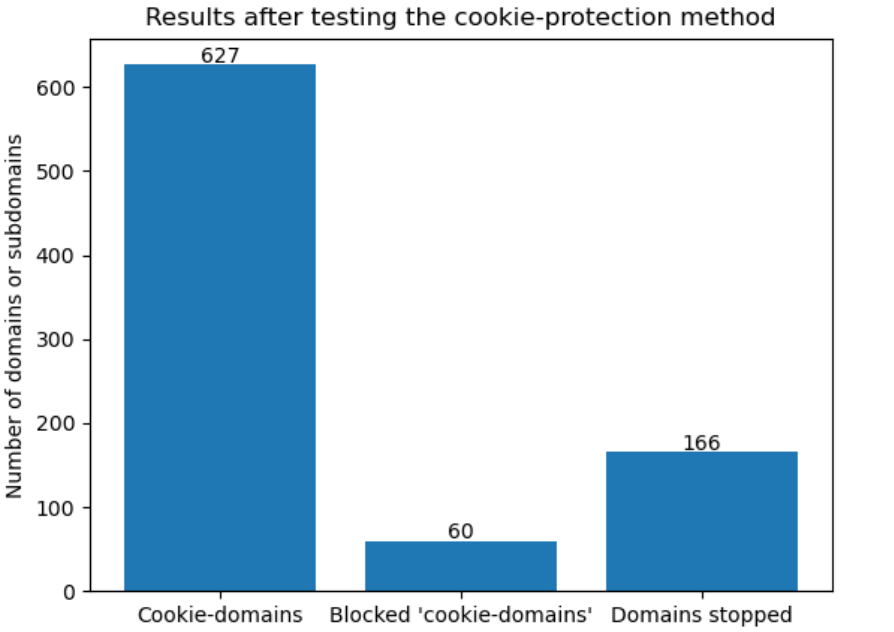


Figure 5.8: Results of testing the cookie-protection approach.

Discussion and Conclusion

During test 1 with my tool, one website completely broke (www.bit.ly). I found out that this website relies on a third-party (cloudfront.net) to provide most of the functionality to the website, such as images, scripts, html and css files. *cloudfront.net* is specified in the *EasyPrivacy* list as a tracker, so this domain is likely used for both purposes. This shows one of the weaknesses by using a blacklist, especially if the blacklist is too strict.

Another weakness with my tool is that cookies used for legit purposes by websites potentially gets removed. For example, cookies from *gstatic.com* were registered as tracking cookies. This domain uses cookies both for advertising/tracking, and to increase performance and reduce bandwidth for Google domains.

One of my assumptions going into the development and implementation of this tool, was that the blacklists wouldn't include as many trackers relying on HTTP cookies as they did. Again, this is because HTTP cookies are so widespread and practically present on every website, and more often than not, used to provide important functionality. However, test 1 and 2 indicates that most domains that utilize HTTP cookies are indeed included in these blacklists, which was a little surprising. This means that the cookie protection approach turns out to be close to redundant, as long as the tool also rely on these blacklists. However, it did work pretty well by itself, stopping tracking attempts by 166 domains, which is close to the result of test 1, with all countermeasures enabled. No websites visibly broke either, which is an improvement from test 1.

Because *EasyPrivacy* broke one website, I suspect that the blacklist is too strict. When it comes to cookies, an improvement could be made to this approach, by finding a way to distinguish between cookies based on their usage, to prevent cookies with legit purposes from being removed. Overall the tool performed well, but I think using a less strict blacklist along with an algorithmic approach to detect more forms of tracking than just HTTP cookies, could yield a better result. The difficulty lies in how to determine whether a web request is made for tracking purposes or not, and where to draw the threshold. As always, it's a trade-off between privacy and usability.

On most websites, the different tools found roughly ($\pm$ 50%) the same amount of trackers. However, I had some interesting results on a couple of websites. For example, on *msn.com*, my tool stopped 17 trackers, Firefox 10 and Ghostery 1. Another example is on *adobe.com*, where my tool stopped 3 trackers and Firefox stopped 30 trackers. I don't have a good conclusion as to why the tools show this big difference on some websites. I also noticed that the amount of blocked trackers can vary from day to day, even on the same website with the same tool. This applies to all the tools. This inconsistency combined with only 30 tested websites can make the results of this test somewhat inaccurate. However, I still think it shows a good indication of the potential of each tool.

It seems that a pure blacklist based approach, like Disconnect use, is a little weak, especially as they only rely on their own blacklists. Ghostery's

supplement of an algorithmic approach, *enhanced anti-tracking*, seems to provide a better result, actually the same as Firefox, that combines blacklist and third-party cookie blocking. Privacy Badger that purely relies on an algorithmic approach, shows that this can potentially indeed compare to the other tools. Brave's combination of multiple methods seems to be the better tool, though, however, a more thorough evaluation should be conducted to produce a more conclusive result.

5.4 Comparison of Blacklists

This section presents an experiment that compares the EasyPrivacy blacklist and the two blacklists provided by Disconnect. It's interesting to see the differences between the two lists, because EasyPrivacy is maintained by the community, while the other lists are maintained by Disconnect in collaboration with Mozilla.

Setup

As mentioned in chapter 4, I combined the two lists provided by Disconnect into one. The source code of the tool is created in a way so one list can easily be enabled while the other is disabled, and the other way around. The experiment consists of having one of them enabled at a time, then running an automated test on the top 100 websites in the Tranco list. In order to automate the test, every domain is read by the tool from the text file and placed in a list. Using the *chrome.tabs* API, it's possible to update the current browser tab to load a new web page. A counter is incremented every time a new page is opened, to grab the next domain from the list containing the top 100 websites. I created a *content script*, that sends a message to *background.js* 10 seconds after a website is loaded, to let *background.js* know it's time to increment the counter, and open the next web page. I specified 10 seconds because it allows the websites to fully load. I define a tracker the same way as in section 5.3 (Definition of one Tracker).

Results and Conclusion

The results can be seen in Figure 5.9. Out of the 100 websites, 6 didn't respond, so the results are from the 94 other websites. EasyPrivacy blocked a total of 607 trackers on these 94 websites, and visibly broke two websites that relied on *cloudfront.net* to provide functionality, as we also saw in section 5.3. The tool blocked tens of requests made to these websites from *cloudfront.net*, but I removed these from the total amount of blocked trackers because it was obvious they were used to provide images and scripts with core website functionality, and were not trackers.

Disconnect's lists showed a decent result, with 489 blocked trackers. This is a good result, considering all the whitelisted domains that are included in this list. No websites visibly broke, which is another positive result.

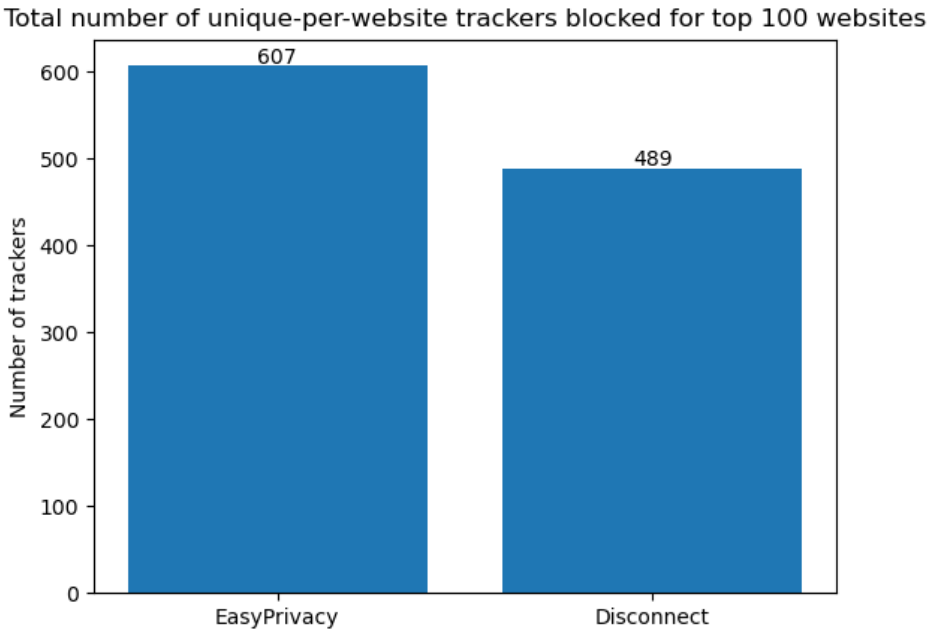


Figure 5.9: Total number of unique-per-website trackers blocked for top 100 websites.

Overall it seems that EasyPrivacy is the better blacklist when it comes to the amount of blocked trackers. But I'm a bit concerned whether the blacklist might be too strict or not, because two websites broke. This list does come with a section of whitelisted domains that can be implemented. This should work similarly to how it works with Disconnect's list. However, the exact domains that provided the functionality that got blocked, doesn't exist in the whitelist section. Therefore, I conclude that a better method for discovering broken websites and potentially whitelist them is needed.

Disconnect and Mozilla seems to be more careful when classifying trackers, as they blocked fewer than EasyPrivacy, probably to limit website breakage, as that is a very important goal. They also seem to more correctly classify tracking domains, and appropriately whitelist domains where needed, if the domain could be used for both functionality and tracking purposes. EasyPrivacy did block more trackers, but choosing which is the overall better list, comes down to how important limiting website breakage is.

5.5 Runtime Evaluation

An important goal of all the tools I have looked at is to reduce the web page load time. This makes sense, as most users wants a faster browsing experience, which makes them more likely to use one of these tools. The

reason why the load time *should* be lower with these tools installed, is because they remove certain web elements on websites from loading. Basically, less things have to load.

Setup

This experiment was conducted on the same 30 websites as the evaluation in section 5.3. I first visited all websites without using any extensions to get a benchmark the other tools in theory should beat. This test, the test for my tool, Ghostery, Disconnect and Privacy Badger were done in Google Chrome.

To precisely measure the page load time I used extensions. In Chrome and Brave I used *Page load time* [96]. This doesn't exist in Firefox, so for that browser I used *app.telemetry Page Speed Monitor* [8]. I visited each page with each tool twice, while clearing the browser cache in between, and between testing each tool, as that influences the load time significantly.
I read that some other factors can influence the time it takes for websites to load, so this is some information about the computer and network where the tests were conducted:

- Desktop running Windows 10.
- Connection: Ethernet.
- Network speed: 80/80 mbps.
- CPU: Intel Core i7-8700k @ 3.70GHz (4.7 GHz Turbo).
- GPU: GeForce GTX 1070 Ti.
- RAM: 16GB DDR4 2666 MHz.

Result

The results of these tests can be seen in Figure 5.10. Default is without any extensions, and the red line makes it visibly easier to see if the other tools are lower or higher than that benchmark.

We can see that the result is indeed a reduction in load time when using these tools compared to using none. An interesting observation is that Privacy Badger is a little bit slower than the others, which might indicate that a pure algorithmic approach makes more calculations and therefore increase the web page load time slightly.

My tool performed worse than the others, which really comes as no surprise. I did not focus on fast and efficient code as that was not the goal of this program. I can definitely see at least one factor as to why it is slower, and that is related to how the blacklists are implemented. When I looked more into the technical details of other tools' implementation, I noticed that they rely on regular expressions to match patterns. This method is faster than adding the patterns to the strings.

Ghostery, Disconnect and Firefox's blacklists are smaller in size than

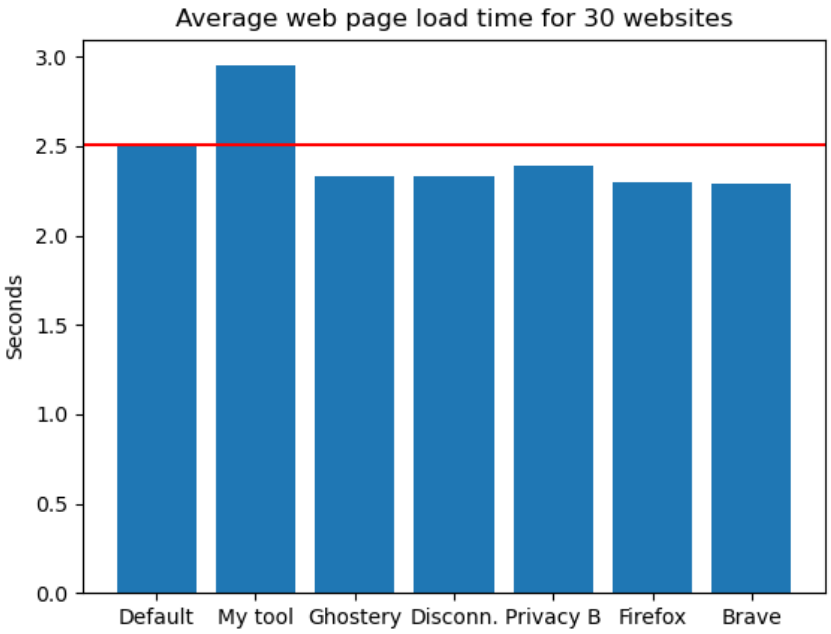


Figure 5.10: Average web page load times on 30 websites in the Tranco list.

the ones implemented in my tool, however, Brave implements several huge lists, so this doesn't seem to be a significant factor when it comes to page load time. Both Brave's and Firefox's protection mechanisms are implemented directly into the browser (compared to the other tools that are implemented as extensions), but it doesn't look like this makes the load times noticeably faster either, as Firefox was on average 0.03 and Brave 0.04 seconds faster than Ghostery and Disconnect.

A weakness with the evaluation is the fact that only 30 websites were visited, and only two visits on the same website with the same tool. Although it does provide some indication as to how the tools perform, a more thorough evaluation could be beneficial.

Chapter 6

Conclusion

6.1 Summary

This research aimed to protect user privacy on the Internet by limiting or blocking web tracking techniques, to prevent them from collecting uniquely identifying information about users and potentially misusing it. In order to understand how this could be done in practice, a literature review were performed to get an overview of the web tracking techniques currently in use. Secondly, countermeasures were researched. Both potential countermeasures proposed by researchers, those already implemented in various tools, as well as certain things users themselves can do. Furthermore, an analysis of the state of the art tools were performed, to understand how these tools limit web tracking in order to protect users' privacy. Privacy oriented browsers were found to have a higher potential to block web tracking than browser extensions, because many tracking techniques rely on access to APIs and browser functions that can be turned off or changed within a browser's source code. In contrast, most protection tools are developed as browser extensions, that have a very limited access to the browser itself, and to the client's local system. Therefore, these tools often rely on blocking lists that contain signatures of known trackers, and intercepts and cancels requests going out to these. This will completely remove all tracking from that resource, but the problem is that the resource needs to be known to be a tracker before it can be classified as such. Some extensions have tried an algorithmic approach, to determine if a resource is a tracker or not, such as Privacy Badger. However, the methods aren't perfect, and previous research has shown it suffers from more false positives than others. In my testing, however, this approach performed just as well as other tools that rely on blacklists.

My main contribution was the development and implementation of a protecting framework to block these web trackers and thereby protecting users' privacy. This tool was implemented as a browser extension in Google Chrome, relying on two different blacklists to block most known web trackers. In addition, an algorithmic approach to detect which domains use third-party cookies, and subsequently strip away cookie-related headers

from HTTP messages to these domains based on their prevalence, was implemented. No other tool I could find rely on both these blacklists as well as using a similar approach to stop cookies.

In order to test how well the tool keep users' privacy protected, it was tested and compared to the state of the art. We saw that my tool blocked more trackers than all the other tools that are implemented as browser extensions, only beaten by the Brave browser. This was a great result, however, I also identified some weaknesses to my tool. First, the *EasyPrivacy* blacklist appeared to be too strict, breaking 1 out of the 30 tested websites. Secondly, cookies used for legit purposes ended up being treated as tracking cookies. Some of these weaknesses could be addressed in future work.

Overall, the tool performed well, and accomplished its goal: by blocking web trackers, users' privacy is protected.

6.2 Research Limitations

This research has potential limitations. First, the *EasyPrivacy* blacklist couldn't be fully implemented. The implementation is missing 0,16% of the entries. This is because *Match Patterns* doesn't allow certain patterns that already existed in the list. Potential ways to overcome this limitation is to use regular expressions when implementing the pattern matching functionality. Overall, though, because as many as 99,84% of the entries were successfully implemented, it most likely wouldn't impact the test results significantly.

Another limitation is that Disconnect's *entities.json* file couldn't be properly fully implemented. It's therefore missing some of the trackers, and potential whitelisted entries. Potential ways to correctly implement these parts, is to have a data structure containing the name of organizations as specified in the list, and their main domain, and check this data structure during the process of reading the file. This way, the function will always know the main domain of every organization. I don't think this would have impacted the test results in any significant way, seeing as most entries in this file were implemented.

Last, the evaluations were conducted on relatively few websites. For a better and more conclusive result, the evaluations should be conducted on, for example, the top 100,000 websites.

6.3 Future work

The approaches implemented in my tool can be improved upon, specifically with regards to the limitations identified in the above chapter. The cookie-protection method can be improved in two ways: (i) by analysing each cookie and identifying what their usage is, to prevent cookies with legit purposes from being removed, and (ii) stopping cookies on a cookie-by-cookie basis, instead of on a domain-by-domain basis. When it comes to the weaknesses of the blacklists, specifically broken websites caused by

EasyPrivacy, there isn't a whole lot that can be done, as we just have to trust that the list is correct.

Future work in this general research field could take several paths. I think researchers should focus on stateless techniques, such as fingerprinting, in the coming years. Seeing as these techniques are easy to use and more transparent to users than stateful techniques, they will likely continue to evolve and grow in use. The best ways to combat these techniques, seems to be by implementing countermeasures directly into browsers, such as Tor and Brave. This gives more flexibility and more options when it comes to potential countermeasures.

When it comes to stateful techniques, much work has been conducted to find and implement countermeasures to these, in particular normal third-party cookies. Because many tracking services (e.g. Google Analytics) now set first-party cookies, future work could attempt to address this issue, without relying on blacklists and without having to block all first-party cookies.

Finally, protecting user privacy from web tracking threats can also be done through policies and regulations, such as GDPR and the ePrivacy Directive, or other systems such as Do Not Track. Researchers and concerned citizens should continue to promote countermeasures like these in the future, to hopefully make them better.

Bibliography

[1] *About Panopticlick*. https://panopticlick.eff.org/about (visited on 18/05/20).

[2] Gunes Acar et al. 'FPDetective: Dusting the web for fingerprinters'. In: Nov. 2013, pp. 1129–1140. DOI: 10.1145/2508859.2516674.

[3] Gunes Acar et al. 'The Web Never Forgets: Persistent Tracking Mechanisms in the Wild'. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. Scottsdale, Arizona, USA: ACM, 2014, pp. 674–689. ISBN: 978-1-4503-2957-6. DOI: 10.1145/2660267.2660347. URL: http://doi.acm.org/10.1145/2660267.2660347.

[4] *Adblock Plus*. https://adblockplus.org/ (visited on 15/04/20).

[5] Gaurav Aggarwal et al. 'An Analysis of Private Browsing Modes in Modern Browsers'. In: Sept. 2010, pp. 79–94.

[6] *Allowing acceptable ads in Adblock Plus*. https://adblockplus.org/acceptable-ads (visited on 25/05/20).

[7] Dennis Anon. *How cookies track you around the web and how to stop them*. https://privacy.net/stop-cookies-tracking/ (visited on 08/06/20).

[8] *app.telemetry Page Speed Monitor*. https://addons.mozilla.org/en-US/firefox/addon/apptelemetry/ (visited on 09/03/20).

[9] *AudioContext - Web APIs*. https://developer.mozilla.org/en-US/docs/Web/API/AudioContext (visited on 04/06/20).

[10] Mika D Ayenson et al. 'Flash Cookies and Privacy II: Now with HTML5 and ETag Respawning (July 29, 2011)'. In: July 2011. URL: https://pdfs.semanticscholar.org/42cf/18892910afd15b0d6872f16384a7bb6cf915.pdf.

[11] Collin M. Barrett. *FilterLists*. https://filterlists.com/ (visited on 21/04/20).

[12] Lilian Besson. *How to use Google Analytics with a beacon image*. https://perso.crans.org/besson/beacon.en.html (visited on 08/06/20).

[13] *BetterPrivacy*. https://betterprivacy.en.softonic.com/ (visited on 15/05/20).

[14] Nataliia Bielova. 'Web Tracking Technologies and Protection Mechanisms'. eng. In: *Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*. CCS '17. ACM, 2017, pp. 2607–2609. ISBN: 9781450349468.

[15] *Brave - How do I use Shields while browsing?* https://support.brave.com/hc/en-us/articles/360022806212-How-do-I-use-Shields-while-browsing- (visited on 20/04/20).

[16] *Brave - Secure, Fast & Private Web Browser*. https://brave.com/ (visited on 15/04/20).

[17] *Brave - The Mounting Cost of Stale Ad Blocking Rules*. https://brave.com/the-mounting-cost-of-stale-ad-blocking-rules/ (visited on 21/04/20).

[18] *Brave | Features*. https://brave.com/features/ (visited on 20/04/20).

[19] *Brave Improves Its Ad-Blocker Performance by 69x with New Engine Implementation in Rust*. https://brave.com/improved-ad-blocker-performance/ (visited on 20/04/20).

[20] Tomasz Bujlow et al. 'A Survey on Web Tracking: Mechanisms, Implications, and Defenses'. eng. In: vol. 105. 8. IEEE, 2017, pp. 1476–1510.

[21] *Canvas API - Web APIs*. https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API (visited on 05/04/20).

[22] *CanvasRenderingContext2D*. https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D (visited on 05/04/20).

[23] *CanvasRenderingContext2D.getImageData()*. https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/getImageData (visited on 05/04/20).

[24] Claude Castelluccia and Arvind Narayanan. 'Privacy considerations of online behavioural tracking'. In: (2012). URL: https://www.enisa.europa.eu/publications/privacy-considerations-of-online-behavioural-tracking.

[25] *Chrome - What are extensions?* https://developer.chrome.com/extensions (visited on 04/05/20).

[26] *Chrome API - chrome.extension*. https://developer.chrome.com/extensions/extension (visited on 05/05/20).

[27] *Chrome API - chrome.runtime*. https://developer.chrome.com/extensions/runtime (visited on 05/05/20).

[28] *Chrome API - chrome.webRequest*. https://developer.chrome.com/extensions/webRequest (visited on 21/04/20).

[29] *Chrome APIs: chrome.storage*. https://developer.chrome.com/apps/storage (visited on 09/06/20).

[30] *Chrome Extension Overview*. https://developer.chrome.com/extensions/overview (visited on 04/05/20).

[31] *Chrome Extensions: Content Scripts*. https://developer.chrome.com/extensions/content_scripts (visited on 09/06/20).

[32] *Chrome Extensions: Getting Started Tutorial*. https://developer.chrome.com/extensions/getstarted (visited on 09/06/20).

[33] *Chrome Extensions: Manage Events with Background Scripts*. https://developer.chrome.com/extensions/background_pages (visited on 09/06/20).

[34] *Chrome Extensions: Manifest File Format*. https://developer.chrome.com/extensions/manifest (visited on 09/06/20).

[35] *Chrome Extensions: Message Passing*. https://developer.chrome.com/extensions/messaging (visited on 09/06/20).

[36] *Chromium Blog - Building a more private web: A path towards making third party cookies obsolete*. https://blog.chromium.org/2020/01/building-more-private-web-path-towards.html (visited on 20/04/20).

[37] *Chromium Web Browser*. https://www.chromium.org/ (visited on 20/04/20).

[38] Catalin Cimpanu. *Flash Used on 5% of All Websites, Down From 28.5% Seven Years Ago*. https://www.bleepingcomputer.com/news/software/flash-used-on-5-percent-of-all-websites-down-from-285-percent-seven-years-ago/ (visited on 13/03/20).

[39] Catalin Cimpanu. *Google to phase out user-agent strings in Chrome*. https://www.zdnet.com/article/google-to-phase-out-user-agent-strings-in-chrome/ (visited on 17/05/20).

[40] *Cliqz*. https://cliqz.com/download (visited on 27/04/20.

[41] John D. Cook. *Supercookies*. https://www.johndcook.com/blog/2019/02/08/supercookies/ (visited on 16/03/20).

[42] *Cookie Header*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cookie (visited on 30/03/20).

[43] *Cookies, Fingerprinting & Co.* https://cliqz.com/en/magazine/cookies-fingerprinting-co-tracking-methods-clearly-explained (visited on 30/03/20).

[44] *Cookies, the GDPR, and the ePrivacy Directive*. https://gdpr.eu/cookies/ (visited on 03/06/20).

[45] Wendy Davis. *KISSmetrics Finalizes Supercookies Settlement*. MediaPost. https://www.mediapost.com/publications/article/191409/kissmetricsfinalizes-supercookies-settlement.html (visited on 17/03/20).

[46] *Detecting User-Agent with JavaScript*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Browser_detection_using_the_user_agent (visited on 03/04/20).

[47] *Developer Chrome - Match Patterns*. https://developer.chrome.com/extensions/match_patterns (visited on 21/04/20).

[48] *Disconnect Blacklists Github*. https://github.com/disconnectme/disconnect-tracking-protection (visited on 16/04/20).

[49] *Disconnect GitHub*. https://github.com/disconnectme/disconnect-tracking-protection/blob/master/descriptions.md (visited on 17/04/20).

[50] *Disconnect Premium*. https://disconnect.me/help\#what-is-disconnect-premium (visited on 16/04/20).

[51] *Disconnect Search*. https://search.disconnect.me/ (visited on 16/04/20).

[52] *Disconnect. (2019)*. https://disconnect.me/ (visited on 15/04/20).

[53] *Do Not Track*. https://www.w3.org/TR/tracking-dnt/ (visited on 23/03/20).

[54] *Document.cookie*. https://developer.mozilla.org/en-US/docs/Web/API/Document/cookie (visited on 30/03/20).

[55] *EasyList*. https://easylist.to/ (visited on 21/04/20).

[56] *ETags*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/ETag (visited on 23/03/20).

[57] *fingerprintjs2*. https://github.com/Valve/fingerprintjs2 (visited on 05/04/20).

[58] *Firefox - Content Blocking. (2019)*. https://support.mozilla.org/en-US/kb/content-blocking (visited on 15/04/20).

[59] *Firefox 72 blocks third-party fingerprinting resources*. https://blog.mozilla.org/security/2020/01/07/firefox-72-fingerprinting/ (visited on 17/04/20).

[60] *Flash - Global Storage Settings Panel*. http://www.macromedia.com/support/documentation/en/flashplayer/help/settings_manager03.html (visited on 15/05/20).

[61] *Flash & The Future of Interactive Content*. https://theblog.adobe.com/adobe-flash-update/ (visited on 07/05/20).

[62] *flash.system Capabilities - AS3*. https://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/system/Capabilities.html (visited on 27/05/20).

[63] *Ghostery - FAQ: How many trackers. (2019)*. https://www.ghostery.com/faqs/many-trackers-ghostery/ (visited on 16/04/20).

[64] *Ghostery - What is Smart Blocking?* https://www.ghostery.com/faqs/what-is-smart-blocking/ (visited on 16/04/20).

[65] *Ghostery Study - Tracking the Trackers. (2019)*. https://www.ghostery.com/study/ (visited on 24/03/20).

[66] *Ghostery. (2019)*. https://www.ghostery.com/ (visited on 15/04/20).

[67] *Google Analytics: Tracking Code Overview*. https://developers.google.com/analytics/resources/concepts/gaConceptsTrackingOverview (visited on 12/06/20).

[68] Y. Haga et al. 'Building a scalable web tracking detection system: Implementation and the empirical study'. In: vol. E100D. 8. Maruzen Co., Ltd., 2017, pp. 1663–1670.

[69] Aniko Hannak et al. 'Measuring Price Discrimination and Steering on E-Commerce Web Sites'. In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. IMC '14. Vancouver, BC, Canada: Association for Computing Machinery, 2014, pp. 305–318. ISBN: 9781450332132. DOI: 10.1145/2663716.2663744. URL: https://doi.org/10.1145/2663716.2663744.

[70] *Header Field Definitions*. https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html (visited on 06/04/20).

[71] *How does Ghostery work?* https://www.ghostery.com/faqs/how-does-ghostery-work/ (visited on 16/04/20).

[72] *How the NSA piggy-backs on third-party trackers*. http://cyberlaw.stanford.edu/publications/how-nsa-piggy-backs-third-party-trackers (visited on 21/03/20).

[73] *HTMLCanvasElement.toDataURL()*. https://developer.mozilla.org/en-US/docs/Web/API/HTMLCanvasElement/toDataURL (visited on 05/04/20).

[74] *HTTP cookies*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies (visited on 30/03/20).

[75] *HTTP Referer header*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referer (visited on 03/04/20).

[76] *HTTP Referer header: privacy and security concerns*. https://developer.mozilla.org/en-US/docs/Web/Security/Referer_header:_privacy_and_security_concerns (visited on 03/04/20).

[77] *HTTPS Everywhere*. https://www.eff.org/https-everywhere (visited on 20/04/20).

[78] *Indexed Database API 2.0*. https://www.w3.org/TR/IndexedDB/ (visited on 18/03/20).

[79] *IndexedDB API*. https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API (visited on 23/03/20).

[80] *ipinfo.io - The Trusted Source for IP Address Data*. https://ipinfo.io/ (visited 29/04/20).

[81] *JavaScript Window Screen*. https://www.w3schools.com/js/js_window_screen.asp (visited on 27/05/20).

[82] Samy Kamkar. *Evercookie*. https://samy.pl/evercookie/ (visited on 21/03/20).

[83] Georgios Kontaxis and Monica Chew. 'Tracking Protection in Firefox For Privacy and Performance'. In: *CoRR* abs/1506.04104 (2015). arXiv: 1506.04104. URL: http://arxiv.org/abs/1506.04104.

[84] Balachander Krishnamurthy, Konstantin Naryshkin and Craig E. Wills. 'Privacy leakage vs. protection measures: the growing disconnect'. In: *In Web 2.0 Workshop on Security and Privacy*. 2011, pp. 2–11.

[85] Pierre Laperdrix, Benoit Baudry and Vikas Mishra. 'FPRandom: Randomizing Core Browser Objects to Break Advanced Device Fingerprinting Techniques'. In: June 2017, pp. 97–114. ISBN: 978-3-319-62104-3. DOI: 10.1007/978-3-319-62105-0_7.

[86] Douglas J. Leith. *Web Browser Privacy: What Do Browsers Say When They Phone Home?* 2020. Available online: https://www.scss.tcd.ie/Doug.Leith/pubs/browser_privacy.pdf.

[87] Pedro Leon et al. 'Why Johnny Can't Opt out: A Usability Evaluation of Tools to Limit Online Behavioral Advertising'. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '12. Austin, Texas, USA: Association for Computing Machinery, 2012, pp. 589–598. ISBN: 9781450310154. DOI: 10.1145/2207676.2207759. URL: https://doi.org/10.1145/2207676.2207759.

[88] J. R. Mayer. *"Any person... a pamphleteer" Internet Anonymity in the Age of Web 2.0*. https://jonathanmayer.org/publications/thesis09.pdf (visited on 26/05/20).

[89] G. Merzdovnik et al. 'Block Me If You Can: A Large-Scale Study of Tracker-Blocking Tools'. In: *2017 IEEE European Symposium on Security and Privacy (EuroS P)*. Apr. 2017, pp. 319–333. DOI: 10.1109/EuroSP.2017.26.

[90] Lou Montulli. *Lou Mountulli and cookies*. http://www.montulli.org/lou (visited on 30/03/20).

[91] Keaton Mowery and Hovav Shacham. *Pixel Perfect: Fingerprinting Canvas in HTML5*. Available online: https://hovav.net/ucsd/dist/canvas.pdf (visited on 05/04/20).

[92] *Navigator - Web APIs*. https://developer.mozilla.org/en-US/docs/Web/API/Navigator (visited on 26/05/20).

[93] *Navigator userAgent Property*. https://www.w3schools.com/jsref/prop_nav_useragent.asp (visited on 17/05/20).

[94] N. Nikiforakis et al. 'Cookieless Monster: Exploring the Ecosystem of Web-Based Device Fingerprinting'. In: *2013 IEEE Symposium on Security and Privacy*. May 2013, pp. 541–555. DOI: 10.1109/SP.2013.43.

[95] Nick Nikiforakis, Wouter Joosen and Benjamin Livshits. 'PriVaricator: Deceiving Fingerprinters with Little White Lies'. In: May 2015, pp. 820–830. DOI: 10.1145/2736277.2741090.

[96] *Page load time*. https://chrome.google.com/webstore/detail/page-load-time/fploionmjgeclbkemipmkogoaohcdbig (visited on 09/03/20).

[97] *Panopticlick - Is your browser safe against tracking?* https://panopticlick.eff.org/ (visited on 18/05/20).

[98] Panagiotis Papadopoulos, Nicolas Kourtellis and Evangelos Markatos. 'Cookie Synchronization: Everything You Always Wanted to Know But Were Afraid to Ask'. In: *The World Wide Web Conference*. WWW '19. San Francisco, CA, USA: Association for Computing Machinery, 2019, pp. 1432–1442. ISBN: 9781450366748. DOI: 10.1145/3308558.3313542. URL: https://doi.org/10.1145/3308558.3313542.

[99] *Privacy Badger*. https://privacybadger.org/ (visited on 15/04/20).

[100] *Public Suffix List*. https://publicsuffix.org/ (visited on 13/03/20).

[101] Franziska Roesner, Tadayoshi Kohno and David Wetherall. 'Detecting and Defending against Third-Party Tracking on the Web'. In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI'12. San Jose, CA: USENIX Association, 2012, p. 12.

[102] *Same-origin policy*. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy (visited on 02/04/20).

[103] Iskander Sanchez-Rola et al. 'The web is watching you: A comprehensive review of web-tracking techniques and countermeasures'. In: *Logic Journal of the IGPL* 25.1 (2017), pp. 18–29. ISSN: 1367-0751.

[104] *Set-Cookie Header*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie (visited on 30/03/20).

[105] *ShareMeNot*. http://sharemenot.cs.washington.edu/ (visited on 16/05/20).

[106] Remy Sharp. *How to detect if a font is installed (only using JavaScript)*. https://remysharp.com/2008/07/08/how-to-detect-if-a-font-is-installed-only-using-javascript/ (visited on 27/05/20).

[107] *Sharpening Our Claws: Teaching Privacy Badger to Fight More Third-Party Trackers*. https://www.eff.org/deeplinks/2019/07/sharpening-our-claws-teaching-privacy-badger-fight-more-third-party-trackers (visited on 17/04/20).

[108] *Silverlight End of Support*. https://support.microsoft.com/en-ca/help/4511036/silverlight-end-of-support (visited on 25/05/20).

[109] Lindsay Simpkins et al. 'A Course Module on Web Tracking and Privacy'. In: *Proceedings of the 2015 Information Security Curriculum Development Conference*. InfoSec '15. Kennesaw, Georgia: ACM, 2015, 10:1–10:7. ISBN: 978-1-4503-4049-6. DOI: 10.1145/2885990.2886000. URL: http://doi.acm.org.ezproxy.uio.no/10.1145/2885990.2886000.

[110] Janice C. Sipior, Burke T. Ward and Ruben A. Mendoza. 'Online Privacy Concerns Associated with Cookies, Flash Cookies, and Web Beacons'. In: *Journal of Internet Commerce* 10.1 (2011), pp. 1–16. DOI: 10.1080/15332861.2011.558454. eprint: https://doi.org/10.1080/15332861.2011.558454. URL: https://doi.org/10.1080/15332861.2011.558454.

[111] Kirk W. Smith. *Privacy Tools Index*. https://www.kwsnet.com/privacy-tools.html (visited 27/04/20).

[112] Christopher Soghoian. *The History of the Do Not Track Header*. http://paranoia.dubfire.net/2011/01/history-of-do-not-track-header.html (visited on 23/03/20).

[113] Ashkan Soltani et al. 'Flash Cookies and Privacy'. In: (Aug. 2009). URL: https://ssrn.com/abstract=1446862.

[114] *SpeedOf.Me API*. https://speedof.me/api.html (visited on 06/04/20).

[115] *Techblog: How We at Cliqz Protect Users from Web Tracking*. https://cliqz.com/en/magazine/how-we-at-cliqz-protect-users-from-web-tracking (visited on 27/04/20).

[116] *The Design and Implementation of the Tor Browser*. https://2019.www.torproject.org/projects/torbrowser/design/\#fingerprinting-linkability (visited on 26/05/20).

[117] *The Facebook Pixel*. https://en-gb.facebook.com/business/learn/facebook-ads-pixel (visited on 12/06/20).

[118] *Tor Project - Browse Privately*. https://www.torproject.org/ (visited on 25/05/20).

[119] *Tracking Prevention in Microsoft Edge (Chromium)*. https://docs.microsoft.com/en-us/microsoft-edge/web-platform/tracking-prevention (visited on 09/06/20).

[120] *Tranco - A Research-Oriented Top Sites Ranking Hardened Against Manipulation*. https://tranco-list.eu/ (visited on 27/04/20).

[121] *UID cookie name*. https://cookiepedia.co.uk/cookies/UID (visited on 02/04/20).

[122] Thomas Unger et al. 'SHPF: Enhancing HTTP(S) session security with browser fingerprinting'. In: Sept. 2013, pp. 255–261. DOI: 10.1109/ARES.2013.33.

[123] *User-Agent*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/User-Agent (visited on 03/04/20).

[124] *Web Storage API*. https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API (visited on 23/03/20).

[125] *WebGL: 2D and 3D graphics for the web*. https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API (visited on 05/04/20).

[126] *What's Brave Done For My Privacy Lately? Episode #2: Third-Party Cosmetic Filtering*. https://brave.com/whats-brave-done-for-my-privacy-lately-episode2/ (visited on 20/04/20).

[127] *What's Brave Done For My Privacy Lately? Episode #3: Fingerprint Randomization*. https://brave.com/whats-brave-done-for-my-privacy-lately-episode3/ (visited on 20/04/20).

[128] *What's Brave Done For My Privacy Lately-Episode #1: Web Resource Replacements (replacing tracking code with privacy-preserving code that keeps sites working well)*. https://brave.com/whats-brave-done-for-my-privacy-lately-episode1/ (visited on 20/04/20).

[129] *Window.navigator - Web APIs*. https://developer.mozilla.org/en-US/docs/Web/API/Window/navigator (visited on 03/04/20).

[130] Akira Yamada, Masanori Hara and Yutaka Miyake. 'Web Tracking Site Detection Based on Temporal Link Analysis and Automatic Blacklist Generation'. eng. In: *Information and Media Technologies* 6.2 (2011), pp. 560–571. ISSN: 1881-0896.

[131] Zhonghao Yu et al. *Tracking the Trackers*. 2016. Available online: https://static.cliqz.com/wp-content/uploads/2016/07/Cliqz-Studie-Tracking-the-Trackers.pdf.