



UNIVERSITAT DE
BARCELONA

Undergraduate thesis

**MATHEMATICS AND COMPUTER
SCIENCE DEGREE**

**Facultat de Matemàtiques i Informàtica
Universitat de Barcelona**

**Online advertisement blocker detection:
A look at the state of the art for
counter-detection and a proof-of-concept
for new approaches**

Author: Joan Bruguera Micó

Director: Lluís Garrido

Submitted to: Department de Matemàtica Aplicada i Anlisi

Barcelona, June 2017

Abstract (English)

In the last years, there has been a surge in the usage of ad-blocking software among Internet users, mainly motivated by intrusive advertisements and rising awareness of the privacy and security implications of advertisements. As a response, many websites have started implementing ad-blocker detection scripts, which detect whether or not an user has an ad-blocking software.

In this work, we study the implications and technical underpinnings of those scripts and document relatively simple workarounds ad-blocking software could implement to avoid detection by those scripts. We also present a prototype implementing the explained techniques over a set of test cases.

Abstract (Català)

En els últims anys, hi ha hagut una explosió en l'ús de software de bloqueig de publicitat per part dels usuaris d'Internet, principalment motivada per la publicitat intrusiva i una creixent consciència de les implicacions de privacitat i seguretat dels anuncis. Com a resposta, moltes pàgines web han començat a implementar scripts detectors de bloquejadors de publicitat, els quals detecten si l'usuari té o no un software de bloqueig de publicitat.

En aquest treball, estudiem les implicacions i principis tècnics d'aquests scripts i documentem mètodes relativament senzills que es podrien implementar en el software de bloqueig de publicitat per tal d'evitar la detecció per aquest tipus d'scripts. També presentem un prototip que implementa les tècniques explicades sobre un conjunt de casos de prova.

Contents

1	Introduction	1
1.1	Structure of the thesis	2
2	Context of advertisement blocking	3
2.1	Internet advertising	3
2.1.1	Basic Internet advertising terminology	3
2.2	Internet advertisement blocking	4
2.2.1	Perceived benefits of ad-blocking	5
2.2.2	Perceived drawbacks of ad-blocking	6
2.3	Reactions to Internet advertisement blocking	6
2.3.1	"Acceptable Ads" and similar initiatives	6
2.3.2	Google Contributor	7
2.3.3	Ad-blocker detectors	7
2.4	Ad-Blocking Software	8
2.4.1	Blacklist-based ad-blockers	9
2.4.2	Heuristic-based ad-blockers	10
2.4.3	Whitelist-based ad-blockers	11
3	Technical principles of ad-blocking software	12
3.1	DOM Model for Web Sites	12
3.2	Ad-blocking Filter Lists	12
3.2.1	Address blocking rules	12
3.2.2	Element hiding rules	13
3.3	How Ad-blocking detection scripts work	13
3.3.1	Address blocking rules	14
3.3.2	Element hiding rules	16
3.4	State of the art of ad-blocking	18
3.4.1	Anti-adblocking detector software	18
3.4.2	Research: 'The Future of Ad Blocking'	19
3.5	Motivation and objectives of the thesis	20
3.5.1	Objective	20
3.5.2	Motivation	20
3.5.3	Overview of the techniques	22
3.5.4	Technical comparison to existing ad-blocking software	24
4	Technical and implementation details	26
4.1	Overview	26
4.2	Initialization process of the server-side component	27
4.3	Network request filtering through proxy	29
4.4	Ad-blocking on the browser	31
4.4.1	Website script context detection	32
4.4.2	Advertiser script context detection	35
4.5	Implementation difficulties	38

4.5.1	IFrames and domain restrictions	38
4.5.2	Various modes to load scripts, images, etc.	39
4.5.3	Content-types and how browsers detect content	39
4.5.4	Performance	39
4.5.5	"Catch-all" solution to performance problems	41
5	Results	42
6	Limitations	43
6.1	Browser support	43
6.2	Detection of the prototype through its 'footprint'	43
6.3	Visual tricks for ad-blocker users	44
7	Conclusions and future directions	45
8	Appendices	46
8.1	Testing the prototype	46
8.1.1	Installing <i>Node.js</i>	46
8.1.2	Obtaining the prototype	46
8.1.3	Installing the dependencies and starting the proxy server	46
8.1.4	Starting the test case server	47
8.1.5	Browse the test cases with a web browser	47
8.1.6	Further steps	49
8.1.7	Test cases	49
8.2	Browser fingerprinting by enabled filter lists	50

1 Introduction

A big proportion of Internet websites nowadays use advertisements as a source of revenue. However, many users consider those advertisements unpleasant, since they are commonly perceived as being distracting or irrelevant, and they have been commonly associated with increased power use and other privacy and security risks.

As a result, **a multitude of advertisement blocker software** (also known as ad-blockers) has been developed, **which aims to block advertisements** (and typically also block other risks, such as malware risks or trackers) to some degree. Such ad-blockers have been adopted by a notable portion of Internet users.

As a reaction to the increasing prevalence of ad-blockers, **some websites have started using ad-blocker detectors**, which are pieces of code that are able to detect whether or not the user has an ad-blocker or not, and change the behavior of the website accordingly to the result. As a further reaction to ad-blocking detection, **various anti-ad-blocker detector techniques and software have been created**, which aim to make a website unable to detect whether an user or not has an ad-blocker installed.

Ad-blocker detectors can exist because **contemporary ad-blockers don't isolate themselves from the website, but rather act on the same internal representation of the website** (the document object model, or DOM) that the websites can access, and thus make their actions easily detectable by scripts acting on the website. On the other hand, **contemporary anti-ad-blocker detectors don't attempt to fundamentally solve this problem**, but rather work by crafting manual workarounds that impede the functioning of single ad-blocker detector implementations.

Recently, there have been early research efforts into fundamentally solving this problem. As recently as April 2017, **a paper explored using APIs that modern web browsers support in order to isolate part of the DOM tree from the website scripts, in order to make their ad-blocker implementation undetectable while masking the ads**. While the techniques given by the researchers effectively achieve this purpose, **there is some room for improvement, since the implementation given by the researchers doesn't have some properties desired by users**, such as altering the website layout when blocking advertisements (instead of masking them), or dealing with privacy and data usage.

In this thesis, **we aim to build on those techniques and propose practical ways to improve on those qualities**.

We explore ways to **allow altering the website layout when blocking advertisements**, by dynamically altering the state the representation of the website when a website script starts or ends execution in order to be able to isolate the changes applies by an ad-blocker. In addition, we also **explore ways to minimize privacy risks** by providing ways to detect and restrict access to privacy-sensitive APIs by advertising/tracking scripts.

In addition, **we provide a prototype implementation of the techniques** we describe, which works under modern web browsers, and we evaluate the results and limitations of those techniques when applied to existing ad-blocker detectors. Finally, we explain the conclusions of our work and consider future directions in this area.

1.1 Structure of the thesis

As for the structure of our thesis, we start by introducing and explaining the current state of Internet advertising: Both the origin and motivation of Internet advertising, the origin and motivation and Internet advertisement blocking, and the reactions to Internet advertisement blocking by website owners and advertisers (see 2).

Next, we explain the technical principles of this kind of software (see 3) that allow it to filter online advertisements, and how ad-blocker detectors work. We also take a look at the existing established advertisement blocking software and its features. To continue, we evaluate the current state of the art of software techniques to avoid detection by ad-blocker detectors and we explain the motivation for proposing a new set of techniques for avoiding detection by ad-blocker detectors, and an overview of their operation.

We then explain in greater detail how we implemented those techniques in an ad-blocker prototype that avoids detection by ad-blockers and the difficulties we have had to deal with during their implementation (see 4).

Finally, we evaluate the results of our ad-blocker prototype against various tests (see 5), and we explain the limitations of our proposed techniques and our implementation of our ad-blocker prototype (see 6). We end by summarizing our results and reaching a conclusion (see 7).

2 Context of advertisement blocking

2.1 Internet advertising

Internet advertising refers to the inclusion of advertising material of any kind, such as text, images, videos, etc., into resource accessible from the Internet, such as websites, emails, apps, etc.

Figure 1: Advertisements on a desktop website. Source: *WordReference*

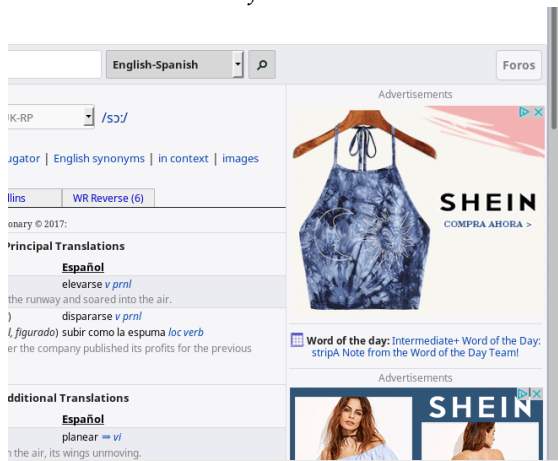
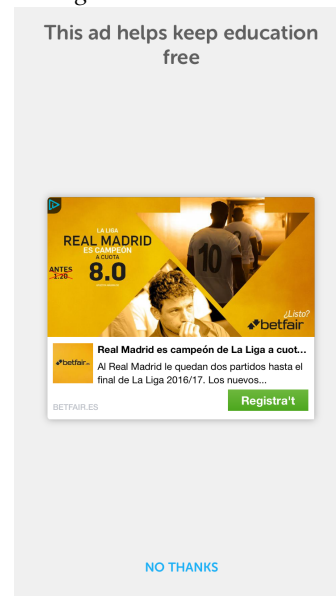


Figure 2: Advertisements on a mobile app. Source: *Duolingo*



Internet advertising is used on a notable proportion of Internet resources, mainly because it provides a revenue source for websites.

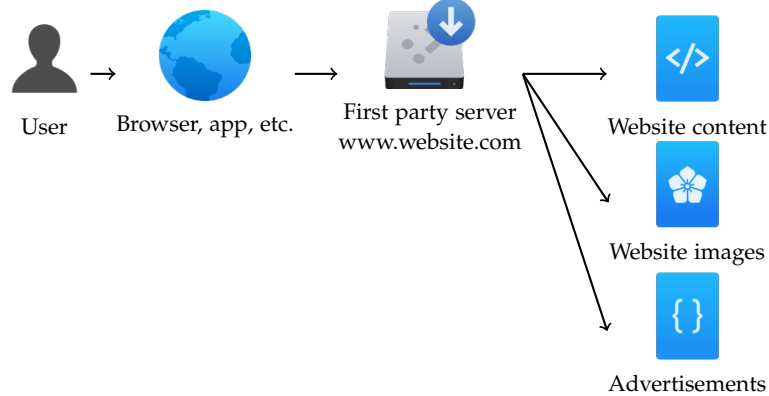
2.1.1 Basic Internet advertising terminology

An internet advertisement is typically called an **ad**, for short.

In the context of Internet advertising, the party that controls the Internet resource where the advertisement is shown is called the **publisher** (such as a newspaper, a social network, etc.). The party that controls the product being advertised is called the **advertiser** (such as a clothing brand, a supermarket, etc.).

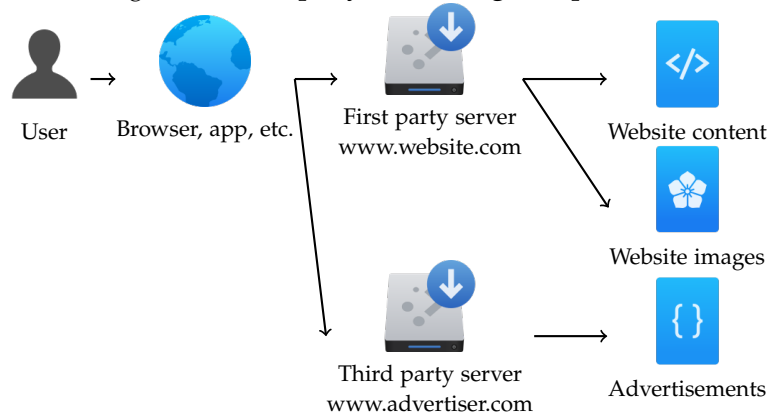
The scheme where the publisher is directly in contact with advertisers, and thus serves the advertisements directly from his website, is called **first-party advertising**.

Figure 3: **First-party advertising (Simplified)**



Otherwise, the scheme where the publisher is not directly in contact with the advertisers, but instead relies on an **advertising network** (such as Google AdSense, AdBlade, etc.) to contact with advertisers and serve advertisements, is called **third-party advertising**.

Figure 4: **Third-party advertising (Simplified)**



Nowadays, the majority of websites rely on third-party advertising via advertising networks, since it avoids having to contact multiple advertisers and simplifies the integration process.

Advertising networks typically provide additional tools. **Tracking** refers to obtaining and storing additional user data, such as relevant interests, browser history, social media profiles, etc.. Tracking enables **targeted advertising** (showing ads based on user data, in order to improve relevancy) and **analytics** (statistical information accessible to publishers about its users).

2.2 Internet advertisement blocking

Internet advertisement blocking refers to any software techniques that are aimed at removing advertisements from web sites and reducing their associated consequences (such

as tracking).

Some basic techniques in this direction are already implemented by default on mainstream browsers (such as *Google Chrome*, *Mozilla Firefox*, *Internet Explorer*, etc.). Examples are pop-up blockers [22], click to play for plugins [36], do-not-track indicators [34], tracking protection [37], etc. However, no mainstream browser implements generic ad-blocking functionality by default. Instead, users must download a separate program or browser add-on, after the installation of which advertisements no longer display on the websites they visit (an "ad-blocker").

Internet advertisement blocking has been soaring in the recent years, and according to various reports by advertising industry members [41] [42], ad-blockers were used by hundreds of millions of Internet users, causing multi-billion dollar losses according to the advertising industry.

2.2.1 Perceived benefits of ad-blocking

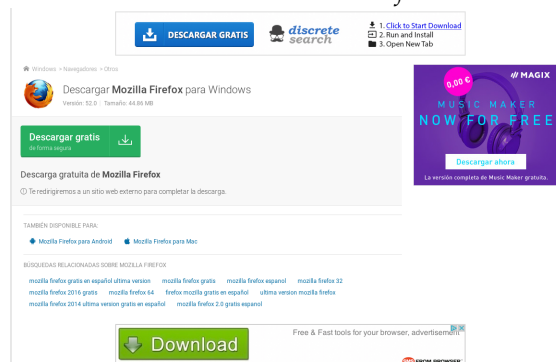
Among the benefits perceived by users of using an advertisement blocker while browsing online, there are [18]:

- **Avoiding irrelevant content:** By blocking advertisements, users can be often able to locate what they are looking for faster in a website, by reducing the amount of visual clutter and content not directly related to their target.
- **Avoiding interruptions and distractions:** Often, advertisements are placed or designed in such a way that they interrupt or otherwise distract the user in a way that impedes him to effectively do their objective. Examples of this are [15]: Excessively big advertisements, advertisements mixed with the content, *pop-ups* [58] and *pop-unders* [58], animated advertisements (such as GIFs or videos), advertisements with sound, interstitial page ads, advertisements disguising as content, pornographic or otherwise inappropriate advertisements, etc.

Figure 5: **Interstitial advertisement:** The user is forced to watch an advertisement and wait before being able to see any content on the website. Source: *Forbes.com*



Figure 6: **Misleading advertisements:** Download buttons for other products before starting a software download. Source: *Softonic.com*



- **Reduced data usage:** If advertisements are blocked, they do not need to be downloaded, and therefore, substantially less bandwidth is used [52]. This is specially important in metered connections, such as most mobile networks.
- **Increased performance and reduced power use:** If advertisements are blocked, no additional computing power needs to be used to download and display them. This can make websites load faster and reduce power use. This is specially important in mobile devices, which are usually limited by battery capacity.
- **Increased privacy:** It has been known that multiple advertising networks attempt to gather and store additional user data in order to do targeted advertising or sell it to other third parties. Examples include: Browser history (cross-website tracking), physical location, IP address, social media profiles, unique user identifiers (such as cookies, local storage, or other 'supercookie'/'fingerprinting' techniques [9]), cross-device tracking, etc.
- **Increased security:** Malware authors have often used advertisements as a way to distribute malware ("malvertising") [60], either by tricking the user into downloading malware programs, or by exploiting browser vulnerabilities in a way that does only require the user to navigate to an otherwise trusted source.
- **Ideological reasons:** Users may wish to block advertisements for ideological reasons, such as avoiding consumerism or avoiding financing inappropriate websites they may incidentally visit (such as hate groups).

2.2.2 Perceived drawbacks of ad-blocking

On the other hand, various publishers and advertisers maintain that advertisements have the following benefits:

- **Revenue source for websites:** Many websites have in-page advertisements as their main or only revenue source. By blocking advertisements, users can decrease the revenue obtained by the website, forcing it to find additional revenue sources (subscriber-only content, crowdfunding, donations, etc.) or closing the website, therefore ultimately reducing the amount of free sources available to users.
- **Advertisements can be relevant:** Some advertisers maintain that their advertisements help users find new products or better offers than they may have found otherwise, therefore benefiting the user.

2.3 Reactions to Internet advertisement blocking

2.3.1 "Acceptable Ads" and similar initiatives

Due to the concerns outlined above, as well as the desire of some ad-blocking software authors to monetize their software, some of the authors of ad-blocking software have started to implement initiatives that aim to find a balance between blocking all advertisements and allowing all forms of advertising, no matter how intrusive it is:

- The ad-blocker software company Eyeo GmbH (the owner of the ad-blocking software *Adblock Plus*) has started an initiative called "Acceptable Ads" [15] [15], which aims to reduce intrusive advertising by defining a set of visual guidelines (placement, size and distinction) that advertisements must follow in order to be displayed. Other restrictions in matters such as tracking or data usage are enforced.

Under this model, advertisements are manually allowed ("whitelisted") by the company behind the ad-blocking software after publishers request and prove their advertisements follow those guidelines. Larger publishers must give the ad-blocking company software and its partners a share of the revenue generating by the advertisements [15] [51]. This has attracted criticism by members of the advertising industry [51] as well as some groups of users [20].

"Acceptable ads" are allowed by default by *Adblock Plus*, but users can opt-out of the initiative and block all advertisements without any penalty.

- Brave software, the company behind the ad-blocking *Brave browser*, has started an "ad-replacement system" [4] that replaces the blocked advertisements by alternative advertisements served by the browser, which they define as faster, safer, and more private [4].

In this system, the revenue generated by the advertisements is split between the users, publishers and the company and its advertisement partners [4]. Publishers must opt-in to the system in order to receive their share. This system has been criticized and is being legally challenged by some publishers [59].

- The authors of other ad-blocking software such as *uBlock Origin* have stated that they have no interest in allowing any advertisement in any form whatsoever by default [56].

2.3.2 Google Contributor

Google Contributor [23] is a program launched by Google offered to remove advertisements from selected websites in the Google advertising platform [53].

The program asks Google users to pay a fee (from 1\$ to 3\$ per month, as decided by the user) to remove ads from the selected websites. Google then pays the selected publishers using the fees paid by the users instead of the revenue generated by the advertisements.

The program launched in late 2014, but was shut down in early 2017, citing a replacement coming in 2017. The replacement program was finally launched on June 2017. [23].

2.3.3 Ad-blocker detectors

Due to the usage of ad-blockers going mainstream (most sources estimate they are used by approximately 10% to 20% of users), many websites have started implementing scripts that detect whether or not an user has an ad-blocker installed and change their behavior according to the results. Various behavior have been attempted, including:

- Blocking access to the user to the website until the ad-blocker is disabled.

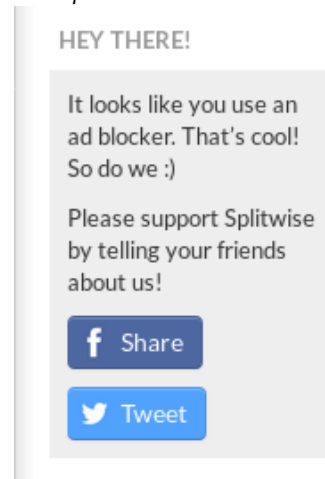
- Asking the user to whitelist the website from its ad-blocker.
- Warning the user that the website depends on advertising revenue, but letting the user continue without further consequences.
- Asking the user to buy a paid subscription to the website.
- Asking the user to help finance the website by using another method (such as donations or crowdfunding).

Those have been met with various degrees of success. Some case studies seem to indicate that blocking access to users until they disable their ad-blockers indeed results in a notable percentage of users disabling it [21] [55], while another notable percentage of users abandon the website.

Figure 7: Ad-blocker detector which blocks users from the website. Source: *Diario de Navarra*



Figure 8: Ad-blocker detector which warns users without blocking them from the website. Source: *Splitwise*



There exist a large amount of ad-blocker detectors, from pre-made scripts such as *Block-AdBlock* [48] to custom solutions developer by publishers.

In the European Union, there was initially some controversy about the legality of using ad-blocker detectors without explicit user consent following the interpretation of the privacy restrictions applied to websites within the European Union [54], but this issue has been settled by the European Union's governing body which has explicitly noted ad-blocker detectors are legal [45].

2.4 Ad-Blocking Software

Here, we explain the main approaches used by ad-blocking software.

Technically, most of the software we refer to as "ad-blockers" is really "content blocking software" - it can be used for purposes other than blocking advertisements, such as blocking malware, inappropriate (pornographic, etc.) content, tracking, social media, etc., without necessarily blocking advertisements. However, we will refer to them as ad-blockers for brevity and in reference to their most common use, blocking advertisements

2.4.1 Blacklist-based ad-blockers

This kind of ad-blockers defined in this list are based on using a list which defines a set of blocking rules (elements and web addresses that should be blocked or hidden when accessing a website). Typically, those lists are manually maintained by volunteers, such as EasyList [11] (explained below).

2.4.1.1 Adblock Plus

Adblock Plus [16] is an ad-blocker that is available for various platforms and browsers (e.g. *Mozilla Firefox*, *Google Chrome*, *Microsoft Edge*, *Safari*). It was one of the earliest ad-blockers available and is currently the overall most popular ad-blocker add-on [1] [8] [16].

Its source code is freely available under the GPL license. Its authors operate a company called EyeO GmbH on Germany who is behind the "Acceptable ads" initiative [15].

2.4.1.2 uBlock Origin

uBlock Origin [24] (also known as *µBlock Origin*, formerly *uBlock*) is an ad-blocker that is delivered as a browser add-on, with wide browser support and similar technical principles to *Adblock Plus*. It is also one of the most popular ad-blocker add-ons [1] [8].

Its source code is freely available under the GPL license. Its author, gorkill (Raymond Hill), does not economically benefit from the project and believes that all advertisements should be blocked with no exceptions [56]. *uBlock Origin* is focused on privacy and performance, with some benchmarks considering it the most performant ad-blocker [25] [46].

2.4.1.3 Brave Browser

Brave Browser [5] is a browser with an integrated ad-blocker. It has taken a novel initiative by creating an "ad-replacement system" [4] that replaces web advertisements with advertisements that are served by the browser, and which are defined by their authors as faster, safer, and more private [4]. The company offers sharing a portion of the revenue generated by the advertisements with the publishers. This system is being legally challenged by some publishers [59].

Its source code is available under the Mozilla Public License.

2.4.1.4 Pi-Hole

Pi-Hole [44] is a network-wide ad-blocker, which is independent of any browser and operating system. It works by creating a DNS server that disallows communication with the domains of third-party advertisers. It is designed to be available to all computers of a network without the need to individually configure each device in it.

The list of blocked third-party advertisers is manually maintained. Due to working at a network level, it has less blocking power than browser add-ons and, for example, is unable to block most first-party advertisements.

Its source code is available under the European Union Public License.

2.4.1.5 AdGuard

AdGuard [3] is a commercial ad-blocker that is available for multiple platforms (iOS, Android, Windows, etc.). *AdGuard for Windows* is capable of working as a man-in-the-middle proxy (a network filter that allows actively intercepting and modifying the content of web requests). This allows *AdGuard for Windows* to work in all programs installed in the computer (all browsers and even desktop applications) without requiring additional configuration for each program, and, unlike DNS solutions like *Pi-Hole*, it maintains the same blocking power as browser add-on based ad-blockers.

Other versions of *AdGuard* for other platforms may have different feature sets according to the platform's limitations. *AdGuard for Windows* is a closed-source program.

2.4.1.6 AdFender

AdGuard [2] is a commercial ad-blocker that is available for Windows. *AdFender* is technically similar to *AdGuard for Windows*. *AdFender* is a closed-source program.

2.4.2 Heuristic-based ad-blockers

2.4.2.1 Privacy Badger

Privacy Badger [12] is a experimental browser add-on focused on blocking third-party advertisers and trackers.

Unlike most ad-blockers which work based on manually defined blocking rules, it automatically detects third-party advertisers and trackers by using an heuristic based on multiple websites referencing the same domain, and thus does not require any manually maintained blacklist. However, due to its heuristic approach, it can easily have false positives (block legitimate domains, such as CDNs), and false negatives (such as advertisers/trackers seen for the first time) [10].

Its source code is available under the GPL license.

2.4.2.2 Visual ad-blocker prototype (Perceptual Ad Highlighter)

Perceptual Ad Highlighter [27] is a prototype ad-blocker based on visual recognition has been developed by a team of Princeton and Stanford University researchers [31] [28]. This kind of ad-blocker doesn't require a manually defined blocking list, but rather visually scans the website looking for content that looks like advertisements. Due to legal requirements on advertising, most advertisements can be easily detected by such a ad-blocker, according to the authors.

Such kind of ad-blocker is in early stages of development and it remains to be seen whether it can be made as reliable as filter list-based blockers. In its current state, it also doesn't address privacy, data usage, or performance related concerns.

2.4.3 Whitelist-based ad-blockers

Another class of ad-blockers are those that require the user to define a list of allowed content. Examples of such class of ad-blockers are NoScript [29] and uMatrix [26].

This kind of ad-blockers generally disallows the download and execution of all kind of content such as images, scripts, etc., and requires the user to manually specify which content is allowed in the browser. This requires effort and technical knowledge on part of the user to define the allowed content, since very often additional content is required for websites to be functional, but its proponents believe the security, data usage and privacy benefits outweigh the disadvantages.

While generally this kind of blockers are not tailored to blocking advertisements, due to their operation method, they block the majority of advertisements and tracking methods by default.

3 Technical principles of ad-blocking software

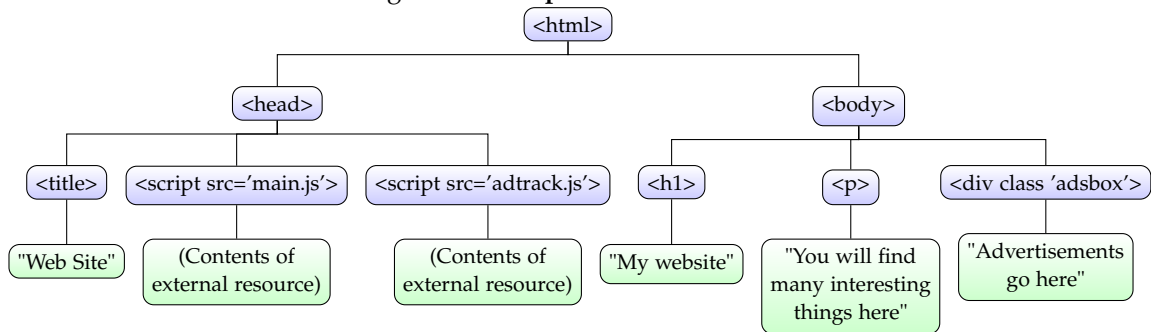
3.1 DOM Model for Web Sites

Websites communicate with the browser to specify the model for the objects in the website using a structure called the DOM (short for 'Document Object Model').

In simple terms, the DOM consists of a tree structure that contains the various objects necessary to display the website, such as the contents of the website (text, images, etc.), special tags that specify how the website should be presented (titles, paragraphs, tables, etc.), and other resources that alter the presentation or behavior of the web site (style sheets, scripts, etc.).

Generally, in web sites, the initial state of the DOM is defined by an HTML document, but the DOM can be modified by other elements (scripts, style sheets, etc.). In addition, elements in the DOM can specify that some resource should be downloaded from an external web address.

Figure 9: Example DOM tree



3.2 Ad-blocking Filter Lists

The majority and the most popular of ad-blocking software in use today works by using what is known as *filter lists*. Filter lists are lists (blacklists) of blocking rules that specify which parts should be blocked from the websites the user visits. Those lists are typically maintained manually by various users and entities.

There are various formats for filter lists with different features. The most popular one, which we will take as a reference, is the Adblock Plus filter list format [17], which is used by the most popular blockers such as *Adblock Plus* [16] and *uBlock Origin* [24] and the most used filter lists such as *EasyList* [11].

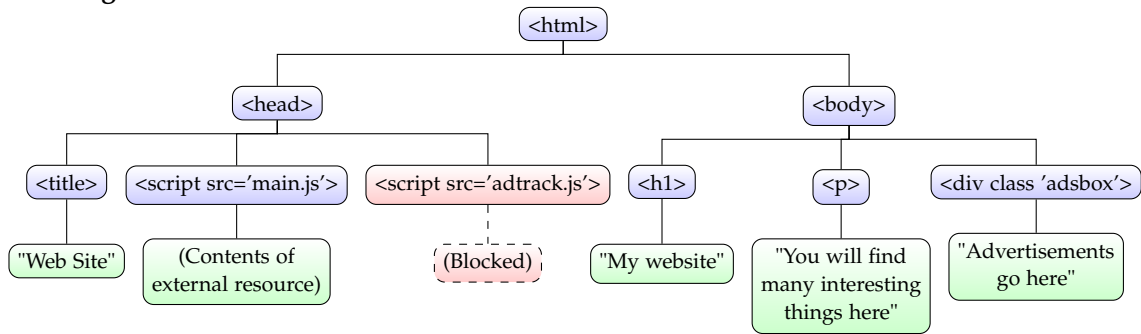
3.2.1 Address blocking rules

Address blocking rules work at the network level, by blocking requests to certain URLs or entire domains.

Due to the usage of advertisement networks by a majority of websites using advertisements nowadays, address blocking rules are often very effective against advertisements.

This is because, to use third-party ads, a website will often include a reference to a script from the third-party domain that will then embed the advertisements in the website. By blocking the requests to the third-party domains, advertisements from those third-parties can be effectively blocked.

Figure 10: Example DOM tree, in which a script resource has been blocked by address blocking rules



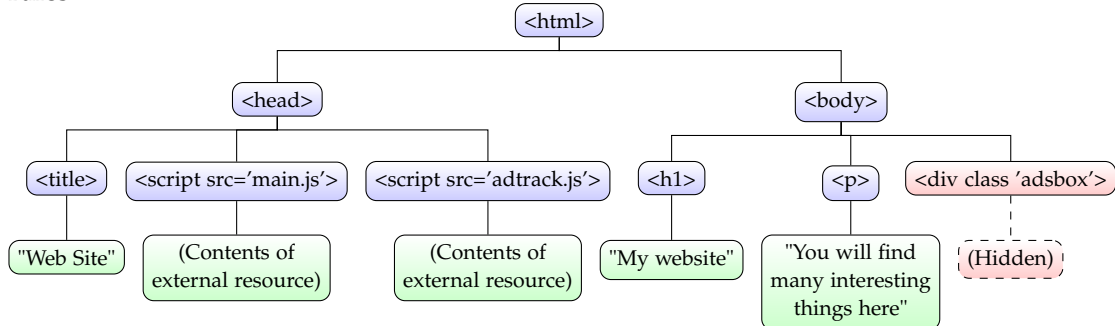
3.2.2 Element hiding rules

Element hiding rules work at the website (DOM) level, by hiding elements in the DOM that are identified as advertisements.

Elements within a website are identified by using CSS Selectors [35]. CSS Selectors allow selecting a website's elements according to various criteria, such as its position within the DOM tree, or one of its attributes (tag name, identifier, class, etc.).

Element hiding rules are typically used for blocking elements that are difficult to block with address blocking rules, such as first-party advertisements, or to remove the elements where third-party ads are embedded (such as *pop-unders* [58], etc.)

Figure 11: Example DOM tree, in which an element has been hidden by element hiding rules



3.3 How Ad-blocking detection scripts work

Ad-blocking detection scripts are JavaScript scripts that are inserted inside a website that check whether or not an ad-blocker is running.

While most popular ad-blockers don't expose any explicit programming interface that allows checking if it is running, web browsers expose many mechanisms that scripts can use to detect if one is running, by checking if known filter rules defined in filter lists are being applied.

A more in-depth analysis of ad-blocking detectors can be seen in the paper *A First Look at Ad-block Detection – A New Arms Race on the Web* [38]

3.3.1 Address blocking rules

One way an script can detect if an ad-blocker is running is by checking whether some requests are being blocked by **address blocking rules**. As a non-exhaustive list, an script can check any of the following to check whether a resource is being blocked by address blocking rules:

- Whether a request to the resource completes successfully.
- Whether the content returned by the request is the expected content.
- Whether a global variable defined in an included filtered script has been defined.
- Whether a type defined in an included filtered script exists.

It's worth noting that detecting ad-blockers by address blocking rules can be unreliable, due to various other factors that can interfere with web requests. For example, factors such as server downtime, unstable wireless networks, network firewalls, etc., could cause a false-positive, i.e. a user without an ad-blocker to be affected.

Example: For example, suppose that a filter list blocks all scripts named `adtrack.js`. Then, one can create a "bait script" named `adtrack.js` that defines a global variable called `VAR` and include that "bait script" in the website.

Then, inside another script (perhaps mixed with the rest of the scripts necessary to display the page), one can check if the `VAR` global variable is defined. If it isn't, then one can consider that the request to the "bait script" has been blocked by an ad-blocker and change the behavior of the page accordingly.

Figure 12: Ad-blocker detection by address blocking, without an ad-blocker

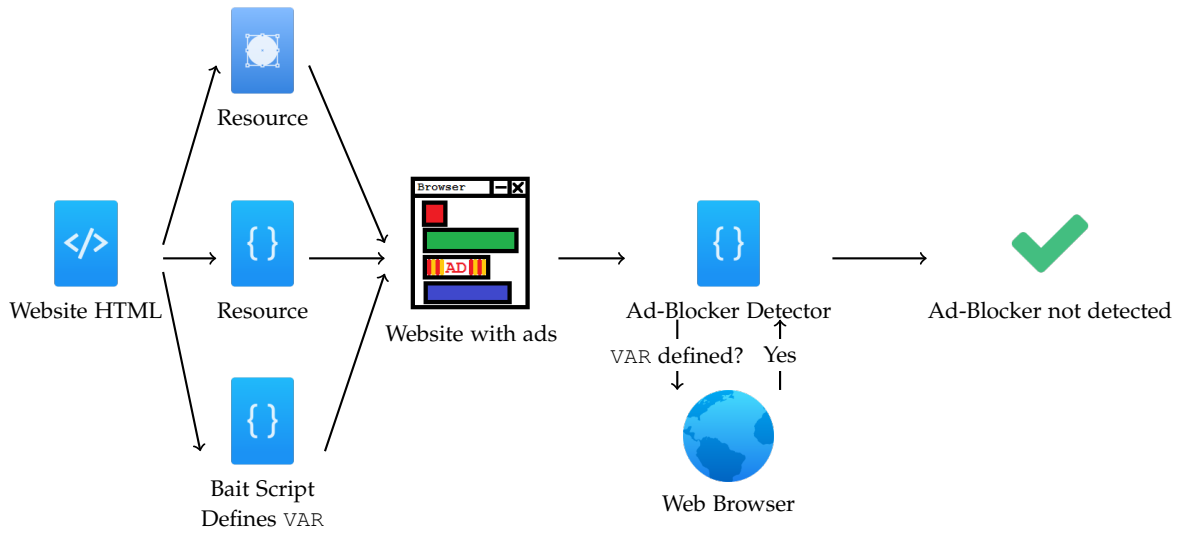


Figure 13: Ad-blocker detection by address blocking, with an ad-blocker

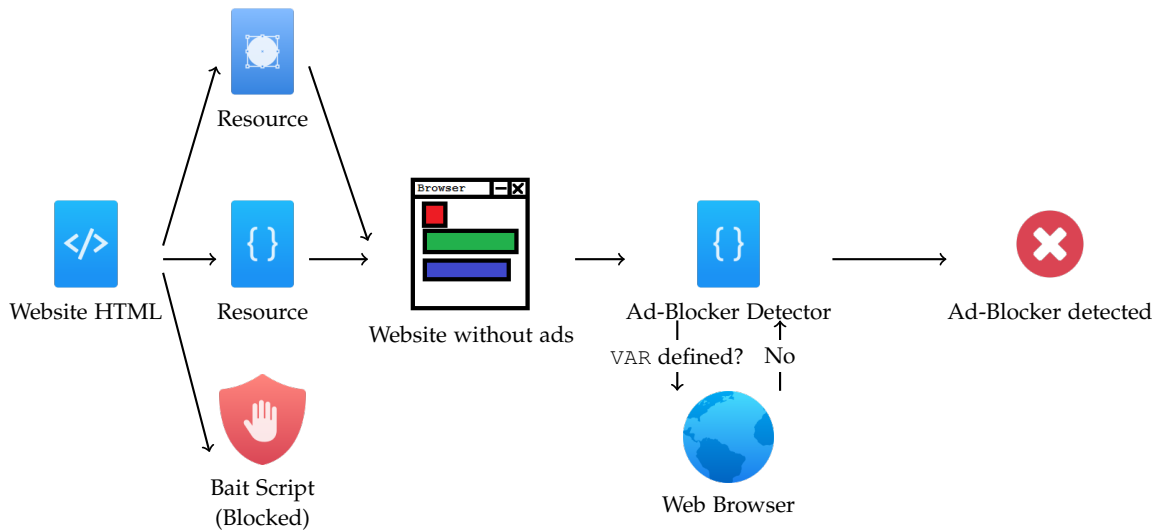


Figure 14: Sample JavaScript code for an ad-blocker detector based on address blocking rules. The address rule 'adtrack.js' is included in the popular *EasyList* filter list, and therefore, that script is not executed under the presence of an ad-blocker.

index.html:

```
1 <script src='adtrack.js'></script>
2 <script>
3   if (window.requestLoaded !== true)
4     alert('Ad-Blocker Detected');
5 </script>
```

adtrack.js:

```
1 var requestLoaded = true;
```

3.3.2 Element hiding rules

Another way a script can detect if an ad-blocker is running is by checking whether some DOM elements are being hidden by **element hiding rules**. As a non-exhaustive list, an script can check any of the following to check whether an element is being blocked by address blocking rules:

- Whether an element is considered visible by the browser.
- Whether an element's height is the expected value (a hidden element is considered to have a height of 0 by the browser)
- Whether the page's elements are positioned as expected (hiding elements can change the page's layout).

Unlike the case with address blocking rules, one can be certain that an ad-blocker or a similar technology is present if detected by such a method, since web browsers have a clearly defined behavior for DOM elements, and there are no other external elements (other than the ad-blocker) that should be able to interfere with it.

Example: For example, suppose that a filter list blocks all DOM elements that have the CSS class `adsbox`. Then, inside an script (perhaps mixed with the rest of the scripts necessary to display the page), one can create a "bait element" assigning the CSS class `adsbox` to it.

After a short period of time (in order to allow the ad-blocker to run), one script can ask the web browser whether the "bait element" is visible or not. If it isn't, then one can consider that the "bait element" has been hidden by an ad-blocker and change the behavior of the page accordingly.

Figure 15: Ad-blocker detection by element hiding, without an ad-blocker

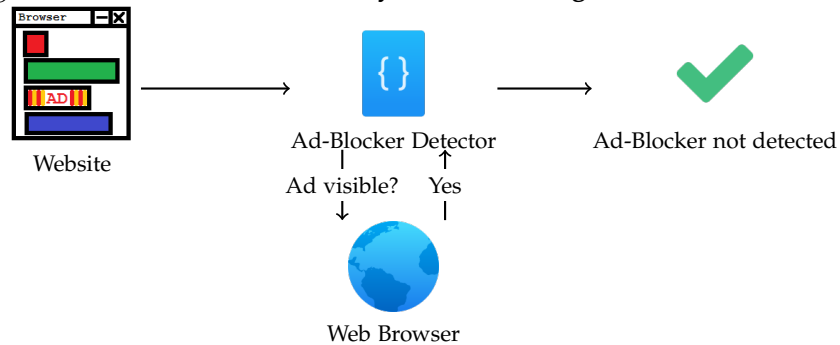


Figure 16: Ad-blocker detection by element hiding, with an ad-blocker

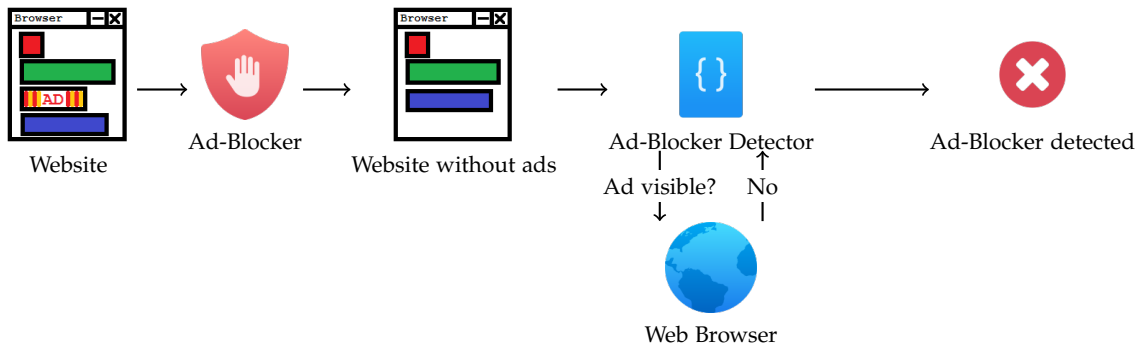


Figure 17: Sample JavaScript code for an ad-blocker detector based on element hiding rules. The CSS class 'adsbox' is included in the popular *EasyList* filter list, and therefore, elements with this class are hidden by the ad-blocker. The usage of `setTimeout` ensures that the ad-blocker has enough time to react.

```

1 <script>
2 window.onload = function () {
3     setTimeout(function () {
4         var baitElement = document.getElementById("baitElement");
5         if (baitElement.clientHeight === 0)
6             alert("Ad-Blocker Detected");
7         baitElement.parentNode.removeChild(baitElement);
8     }, 500);
9 };
10 </script>
11 <div class="adsbox" id="baitElement">BAIT</div>

```

3.4 State of the art of ad-blocking

3.4.1 Anti-adblocking detector software

3.4.1.1 AdBlock Plus, uBlock Origin (and similar techniques by ad-blockers)

AdBlock Plus expanded the definition of the AdBlock Plus filter list format to include two new options aimed at avoiding detection by some of the most simple techniques used by ad-blocker detectors [19]. Those new options are supported by most blacklist-based ad-blockers. However, ad-block detectors can easily use other available techniques that are not affected by these new options.

uBlock Origin includes various lists (under the name "uBlock filters"), which are enabled by default, that include workarounds for generic ad-blocker detectors such as *Block-AdBlock* [48] or specific workarounds for popular websites. Additionally, it offers installing anti-ad-blocker detector lists such as those of *AdBlock Protector* [33] or *Anti-AdBlock Killer* [47].

3.4.1.2 Anti-Adblock Killer

Anti-Adblock Killer [47] is a set consisting of a blacklist and a browser add-on (in the form of a user script) that is designed to work along other ad-blockers (such as *AdBlock Plus* or *uBlock Origin*) and which tricks ad-blocker detection scripts and other countermeasures into thinking that no ad-blocker is installed.

Technically, it works by various means (blocking URLs, replacing website scripts, adding elements to the page, etc.), all of which require manually understanding the way each website detects an ad-blocker and implementing a workaround to avoid the detection. As such, it is not a generic anti-ad-blocker detector.

Its source code is available under a Creative Commons BY-SA license [47].

3.4.1.3 AdBlock Protector

AdBlock Protector [33] is a solution which is technically similar to *Anti-Adblock Killer*. Its source code is available under the GPL license [33].

3.4.1.4 Visual ad-blocker prototype (Perceptual Ad Highlighter)

Perceptual Ad Highlighter [27] (previously covered) also pioneers a generic anti-ad-blocking detector technique. Unlike most ad-blockers, *Perceptual Ad Highlighter* doesn't completely hide the advertisements from the page, but rather covers them with a layer marking them as advertisements.

In order to avoid detection by ad-blocker detectors, *Perceptual Ad Highlighter* uses the capability of modern web browsers to overwrite the implementation of the APIs used by websites to examine the contents of the website. It uses this technique to prevent the website from being able to enumerate or interact with the layers covering the advertisements. The techniques used by this software are covered further in later sections.

3.4.2 Research: 'The Future of Ad Blocking'

During the elaboration of this thesis, a paper called *The Future of Ad Blocking: An Analytical Framework and New Techniques* [28] was published that explores similar techniques to the ones we will explore in this thesis. The following is a brief summary of the techniques explained in the paper:

- The paper argues that ad-blocking, ad-blocking detection and anti-ad-blocking detection game isn't a never-ending arms race, but rather a situation that ad-blocker users are fated to win.

A simplified explanation of the argument is that the only reason that ad-blocker detectors can exist nowadays is because the current generation of ad-blockers act on the same level as websites (due to working on the same DOM tree accessible to the website's scripts), which renders them detectable. However, they note that since ad-blockers can work at a higher privilege level than websites, it's possible for them to completely hide their changes to the website's DOM tree, rendering them undetectable. Additionally, they note that current browsers already expose certain APIs that make this approach possible.

- The paper explores using visual detection for advertisement detection, unlike the common mainstream manually maintained blacklist approach. They report that they are able to achieve good accuracy, mostly due to the fact that most advertisements on popular websites have distinguishing tells that make them recognizable.

Figure 18: As an example, the AdChoices logo is present on many advertisements on popular websites, which makes them easily detectable by visual recognition

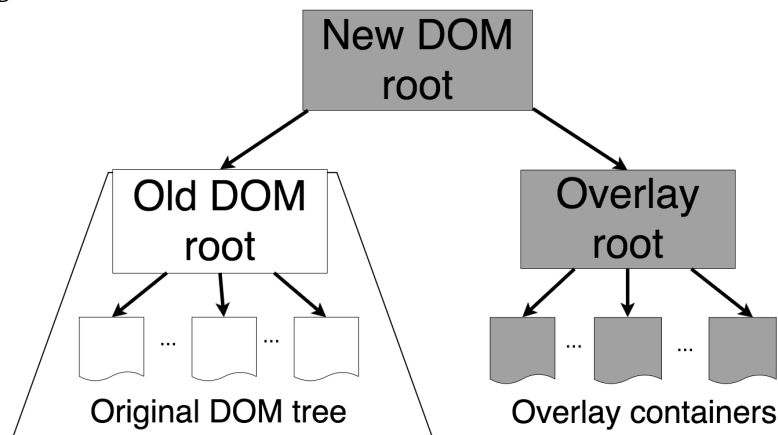


- The paper explores avoiding detection by ad-blocker detector scripts by using APIs implemented in most modern browsers (Firefox, Chrome, Edge, etc.), that allow overwriting the browser's native DOM properties with user-defined functions.

The author's approach is to make part of the DOM inaccessible to the website scripts. In this hidden part of the DOM, elements overlaying the advertisements are placed in order to mask the ads.

This effectively avoids detection by ad-blocker detectors, since it creates a 'safe' area in the DOM where the elements overlaying the advertisements can be placed, without the website being able to detect them as they are unable to access this area of the DOM.

Figure 19: Model presented by the researchers. They separate the website's DOM ("Old DOM root") from the part of the DOM being used to mask the advertisements ("Overlay root") and make it inaccessible by the website's scripts. Source: 'The Future of Ad Blocking'



- Due to its visual approach, the paper doesn't outline how to maintain the privacy (tracking), data usage and performance considerations. Since the approach of the paper doesn't rely on filter lists but rather on visual detection of advertisements, those issues can't be properly addressed, because a tracking script can be implemented without showing anything visually, and the advertisement resources should be downloaded for visual detection to happen.
- The paper presents a prototype implementation of their techniques as a Chrome browser extension, called *Perceptual Ad Highlighter* [27].

3.5 Motivation and objectives of the thesis

3.5.1 Objective

In this paper, we propose modifications to blacklist-based ad-blockers, that add the capability to avoid detection by ad-blocking detectors in a generic way that doesn't require manual workarounds for each website (unlike software like *Anti Ad-Block Killer* [47]).

In comparison with similar techniques presented in *The Future of Ad Blocking* [28], we show that those can be implemented in a way that allows removing the space used by the advertisements in the page, and compromising to a lesser degree the privacy-enhancing and data-usage features of ad-blockers.

3.5.2 Motivation

3.5.2.1 Educational

It is of research interest to study the principles under which Internet advertising, ad-blockers and ad-blocker detectors work, and which are the ways they can be avoided,

in order to be able to provide accurate information to users of the implications of those technologies.

3.5.2.2 Website Compatibility

In most circumstances, it is desirable to avoid website scripts from being able to detect the changes produced by an ad-blocker, because this minimizes the risk of the website scripts malfunctioning in the presence of an ad-blocker.

In addition, the techniques we use to avoid detection by ad-blockers can also be used to implement arbitrary aesthetic changes to websites without breaking compatibility with the website scripts.

3.5.2.3 Malvertising and privacy risks

Since advertisements are often used as a vector for distributing malware [60], in some environments it may be not be considered desirable to disable ad-blockers in order to access certain websites, in order to maintain protection against malware and privacy risks.

3.5.2.4 Avoiding annoyances and content restrictions

Like advertisements, ad-blocker detectors and its associated behaviors (prompts to disable ad-blockers, avoiding access to applications or websites, etc.) are often considered an annoyance by users. Users who have a clear stance about advertising (e.g. only acceptable ads, or no advertisements at all), may wish to avoid those restrictions to the maximum possible extent.

3.5.2.5 Implementing "Acceptable ads" and ad replacement plans

Ad-blocker detection and its associated behaviors may hinder adoption of alternate advertisement policies such as "Acceptable ads" [15] or ad-replacement policies [5] by discouraging users to use ad-blockers, therefore impacting the adoption of measures that may improve the overall Internet advertising experience for users.

3.5.2.6 Fingerprinting

Browser fingerprinting is a method of tracking web browsers that relies on identifying an internet user by the settings and unique characteristics of his browser, such as the browser, operating system, fonts, language, plugins, etc., (a "fingerprint") instead of traditional methods such as cookies, local storage, IP addresses, etc. [13] [50].

Unlike the traditional methods of tracking, which are mostly transient and easily cleaned by standard browser utilities, it is very difficult to defend against fingerprinting, both because it's difficult to detect when it's being used, and because the characteristics of the fingerprint are hard to change.

Whether or not a user uses an ad-blocker can be used as a characteristic of the fingerprint (as detected by an ad-blocker detector). In addition, in the annexes (see 8.2), we show that it's possible to detect which filter lists an user has enabled, which creates a much

more accurate and persistent fingerprint (multiple bits of fingerprint information instead of a single bit). Therefore, it is of interest to forbid detection by ad-blocker detectors in order to improve online privacy.

3.5.2.7 Tracing / Statistics

While not the main aim of this thesis, the techniques we present to detect when an advertiser script is being executed and when it accesses privacy-sensitive browser APIs can be used in order to gather information about the privacy consequences of those scripts without requiring manual analysis or reverse engineering.

3.5.2.8 Blocking of advertisements in locked-down contexts

Our implementation of an ad-blocker as a main-in-the-middle proxy is technically capable, unlike other ad-blockers, of blocking advertisements in browsers and mobile applications that are traditionally not blocked by network-level solutions such as *Pi-Hole* [44] that only work at the network-level by blocking access to certain hosts.

While similar implementations to our ad-blocker prototype in this sense exist, we have not found any open source implementation that work in such manner.

3.5.3 Overview of the techniques

We propose the following two techniques that could be added to existing blacklist-based ad-blockers, which enable reliable generic ad-block detector avoidance:

- **Website script context detection:** We examine how to implement a way for ad-blockers to receive a callback when a script in a website is entered or left. This can be used by ad-blockers (or any similar program, such as other browser extensions), to restore the website to the state ad-blocker detectors expect when executing (for example, virtually redisplaying the advertisements when a website script is entered, and hiding them after the website script execution finishes).

This technique is mostly concerned with avoiding detection by ad-blocker detectors that work by abusing **element hiding rules**.

This is accomplished by either automatically inserting hooks to the contents of the website scripts, or by a technique based on overwriting certain DOM properties.

Unlike in *The Future of Ad Blocking: An Analytical Framework and New Techniques* [28], this method doesn't require hiding part of the DOM, but rather allows the ad-blocker to work in a way that is more similar to the way current mainstream ad-blockers work (which don't add overlay elements to mask the ads, but rather completely hide the ads, visually removing them from the website layout).

- **Advertiser script context detection:** In order to avoid detection by those ad-blocker detectors based on **address blocking rules** in a generic way, we necessarily have to allow the advertiser's resources to be downloaded and executed in order to effectively avoid detection by ad-blocker detectors, since otherwise it's trivial to build an ad-blocker detector using a sufficiently complex script. Therefore, we examine how

to allow those scripts to execute while simultaneously avoiding access to privacy-sensitive APIs (e.g. cookies, location, mouse position, etc.) by advertiser scripts (i.e. those scripts that are filtered by filter lists). However, regular website scripts should be able to access those APIs normally, since websites typically depend on those for proper function (e.g. cookies for persistent authentication to websites, location for maps applications, etc.)

We propose two approaches, which are analogous to the ones used for website script context detection. One approach is modifying the advertiser scripts to insert hooks in order to receive a callback when an advertiser script is entered or left. Another approach is detecting when an advertiser script accesses a privacy-sensitive browser API by overwriting it (similarly to overwriting certain DOM properties) and using non-standard but widely supported browser APIs in order to walk the call stack and detect if an advertiser script has caused the call to occur or not.

When an advertiser script accesses a privacy-sensitive API, we can present a 'virtualized' version of the API that avoids exposing sensitive information. For example, the cookies set by the advertiser script can be non-permanent and cleaned when the user exits the website, or the access to location APIs can return falsified information.

However, it's worth noting that, when implementing **advertiser script context detection**, we reintroduce the **additional data usage** caused by downloading the resources required by the advertisements, which in classical blacklist-based ad-blockers are blocked. In addition, as we will see, applying those two techniques will have a non-negligible performance impact.

To mitigate those problems, we propose (but don't implement in our prototype) a third modification to blacklist-based ad-blockers. This modification would be to add an additional filter option in filter lists, which would determine whether to apply the proposed techniques. By default, an ad-blocker would not apply the proposed techniques, and therefore would work like the contemporary ad-blockers. However, if such a filter option were present, the anti-ad-blocker detection techniques would be applied.

Both under the observation that nowadays, only a relatively small proportion of websites employ ad-blocker detectors, and the game-theoretical observations in *The Future of Ad-Blocking* [28] (publishers and advertisers have no incentive to use ad-blocker detectors if effective anti-ad-blocker detectors are available), we expect that such an option would effectively balance the desire to avoid additional data usage with the desire to use anti-ad-blocker detectors.

It's also worth noting that we have implemented the proposed ad-blocker as regular scripts which are inserted into the website **by a man-in-the-middle proxy** (similarly to *AdGuard* or *AdFender*), instead of a browser extension (similarly to *AdBlock Plus* or *uBlock Origin*).

This has the advantage of browser independence, requiring only a setup for the whole network, and allowing the implementation to work even in "locked-down" contexts such as mobile browsers or mobile apps. Due to the increasing prevalence of those contexts, this is an approach to implementing ad-blockers that is increasingly valuable.

However, we also note that this is an implementation detail is orthogonal to all other features of ad-blockers (i.e. any ad-blocker can be implemented in this way), and therefore isn't an essential part of our prototype.

3.5.4 Technical comparison to existing ad-blocking software

The following table summarizes the capacity of the various ad-blockers in the market and research prototypes, according to the explained balances of features:

Figure 20: **Our interpretation of the various relative strenghts and weaknesses of ad-blockers, according to various criteria**

Ad-Blocker	Blocks ads	Privacy	Undetectable	Data Usage	Reliable	Maintenance
(No ad-blocker)	No	No	N/A	No	N/A	None
Group 1 (Classic blacklist-based)						
AdBlock Plus	Yes	Yes	No	Yes	Yes	Filter lists
uBlock Origin	Yes	Yes	No	Yes	Yes	Filter lists
Brave Browser	Yes	Yes	No	Yes	Yes	Filter lists
Pi-Hole	Only 3rd-party	Yes	No	Yes	Yes	Host lists
AdGuard	Yes	Yes	No	Yes	Yes	Filter lists
AdFender	Yes	Yes	No	Yes	Yes	Filter lists
Group 2 (Classic anti-ad-blocker detectors along with a supported ad-blocker)						
Anti-Adblock Killer	Yes	Yes	Yes	Yes	Yes	Filter lists + detector fixes
AdBlock Protector	Yes	Yes	Yes	Yes	Yes	Filter lists + detector fixes
Group 3 (Heuristic-based)						
Privacy Badger	Only 3rd-party	Yes	No	Yes	No	None
Perceptual ad hi.	Yes	No	Yes	No	Mostly yes	Visual acc.
Group 4 (Presented prototype)						
Content-Filter-Policy	Yes	Yes (Ideally)	Yes	Mostly no	Yes	Filter lists
Group 5 (Technically possible implementation with improved filter lists)						
(Not implemented)	Yes	Yes	Yes	Mostly yes	Yes	Filter lists + undetectable option

On the first group, there are the **classical blacklist-based ad-blockers**, which work by simply blocking addresses/elements according to blocking rules. Those can generally block all kinds of advertisements, improve privacy and reduce data usage. However, due to their implementation, they can be easily detected by ad-blocker detectors, and require manual maintenance of the associated filter lists.

On the second group are the **ad-blockers from the previous group, combined with a classic anti-ad-blocker detector** based on manual workarounds. Those add protection from ad-blocking detectors, at the cost of adding further maintenance costs.

On the third group are the **experimental heuristic blockers**. *Privacy Badger*, due to its heuristic approach to detecting third-parties, isn't too reliable and can't block all kinds of advertisements, in addition to not having any anti-ad-blocker detector techniques. *Perceptual Ad Highlighter* uses visual detection to detect advertisements, but due to its approach,

it can't address the privacy and data usage problems, and while it can reliably block most advertisements, is constrained by the accuracy of the visual detection.

On the fourth group is **our proposal**, which is a modification to the classical blacklist-based ad-blockers that adds anti-ad-blocker detector techniques. This comes at the cost of incremented data usage, but doesn't add any additional maintenance costs to blacklist-based ad-blockers. Completely locking down access to privacy-sensitive APIs requires work due to their extent, but is possible with enough effort.

On the fifth group is **a technically possible (but not implemented) proposal**, which is a modification of our proposal that aims to improve data usage. Under this proposal, a filter option would be added to filter lists that would specify whether to block advertiser scripts (necessary for data usage) or allow them in a restricted environment (necessary for avoiding detection by ad-blocker detectors). This would address data usage concerns with a negligible impact on filter list maintenance.

4 Technical and implementation details

4.1 Overview

As we have noted in the previous sections, our ad-blocker prototype is based on injecting an ad-blocker into the websites as they are downloaded, by the means of a man-in-the-middle proxy that intercepts the requests and inserts the ad-blocker into the code that is interpreted by the browser.

As such, our prototype can be split in two parts: A **server-side component** that executes as a standalone application that prepares the ad-blocker component, and starts the proxy, and inserts the ad-blocker into the website requests, and a **client-side component** that is executed inside the browser that blocks the advertisement and in addition executes the proposed techniques.

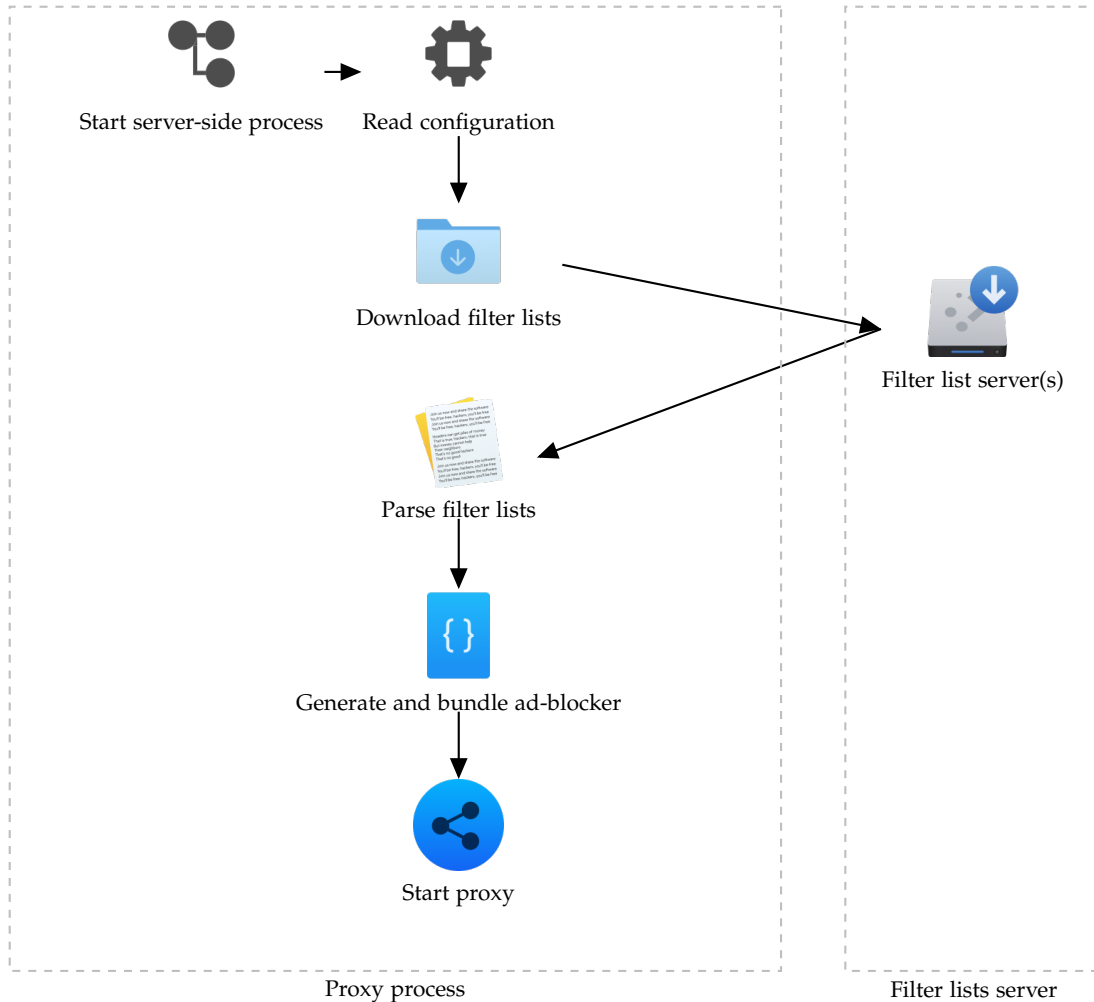
We have used JavaScript for both our server-side and client-side components, since JavaScript is the most popular language for web-related development and developing both components in this language facilitates reuse between our server-side and client-side components. To implement the server-side component, we have used *Node.js* [39], which is a JavaScript execution environment based on Google Chrome's V8 JavaScript execution engine, which enables development of JavaScript applications (e.g. command line applications, desktop applications, web servers, etc.) outside a browser environment. In order to import third-party libraries, we used *npm (Node Package Manager)* [40], which is a package manager that is installed alongside *Node.js* and provides access to the largest collection of JavaScript packages.

The functionality of our ad-blocker prototype implementation can be broken in three phases: **Initialization**, **Network request filtering** and **Ad-blocking in the browser** (including **Website script context detection** and **Advertiser script context detection**), which we detail below.

4.2 Initialization process of the server-side component

This diagram explains the initialization process of the ad-blocker prototype:

Figure 21: Initialization process for the prototype



- We start by **reading a configuration file** which is specified in JSON format, which contains the various parameters of the ad-blocker, such as: How to filter the advertisements (hide them, mark them, etc.), which privacy-sensitive APIs to override, and which filter lists to use to filter advertisements, etc.
- We then **download the filter lists** which are specified in the configuration file. The filter lists should be specified in the Adblock Plus filter list format [17], the most common filter list format which is used by the most popular lists such as *EasyList* [11].
- We then **parse the filter lists** we have downloaded, in order to be able to use them

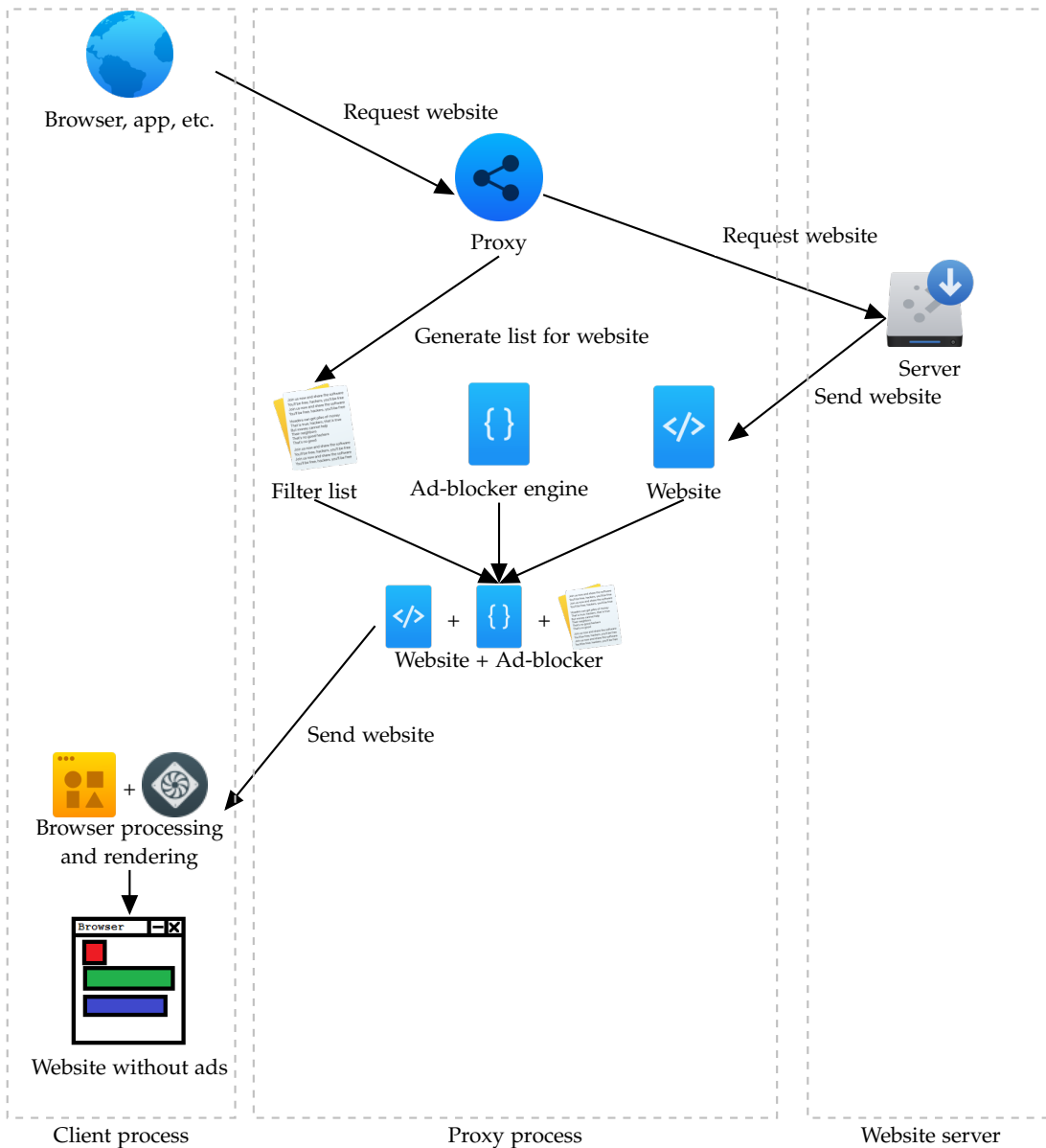
later in our client-side ad-blocker component. In order to parse the filter lists in AdBlock Plus filter list format [17], we used an existing library called *abp-filter-parser* [6], which can parse and provide the list of AdBlock Plus filters that should be applied to a certain website. It was used as the basis for a C++ library that was later adopted by the *Brave browser* for its ad-blocking feature.

- We then **generate and bundle the ad-blocker engine** along with all the libraries and additional code that it requires, into a single-file JavaScript package which can be easily delivered to the web browser. We have used the *browserify* [7] library for this, which facilitates reuse between our server-side component and our client-side component.
- Finally, we **start the man-in-the-middle proxy server**, which will intercept incoming requests and insert the ad-blocker component into them. We have used the *http-mitm-proxy* [32] library, which can set up a man-in-the-middle proxy that allows us to intercept incoming requests and insert our ad-blocker inside them.

4.3 Network request filtering through proxy

This diagram explains the network request filtering process by the proxy server of the ad-blocker prototype:

Figure 22: Diagram of the mechanism used by the prototype to inject the ad-blocker into the browser



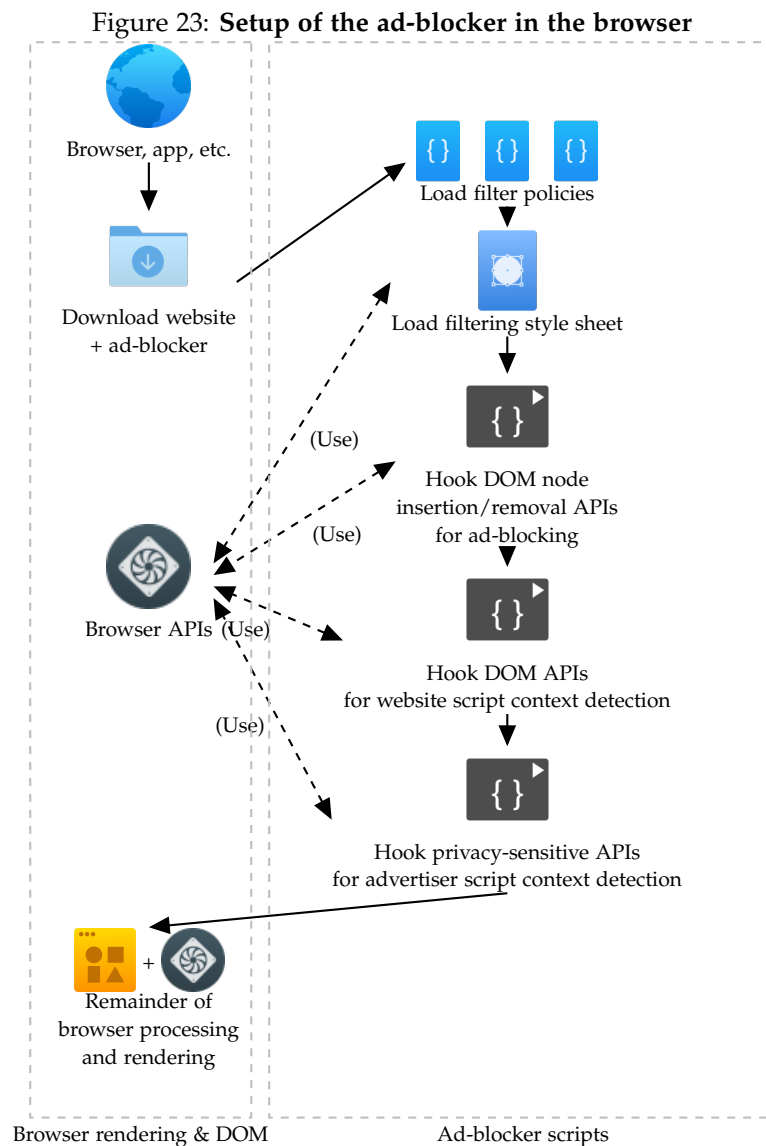
- First, the network request from the web browser (or other Internet application) is **intercepted by the proxy**.

- The proxy, then, **forwards the request** to the original web server and receives the original website.
- At the same time, a **list of ad-blocking filters that apply to the website is generated**.
- Before returning the response received from the website server, both the bundled **ad-blocker engine and the filter lists are injected into the start of the response**.
- When this response is received by the web browser, **the website is executed along with the ad-blocker prototype code**, which ultimately causes the website to be rendered without advertisements.

4.4 Ad-blocking on the browser

After the browser finishes the website request, which is intercepted by the proxy and manipulated to insert the ad-blocker, the browser starts execution of the received contents. The ad-blocker code is the first element to be processed by the browser, which sets up the necessary events.

Next, we show a small diagram that shows how the ad-blocker works in the context of our proposed techniques. However, we will avoid a detailed explanation of the ad-blocker part of our prototype, since it generally works similarly to the existing classic blacklist-based ad-blockers that already exist on the market.

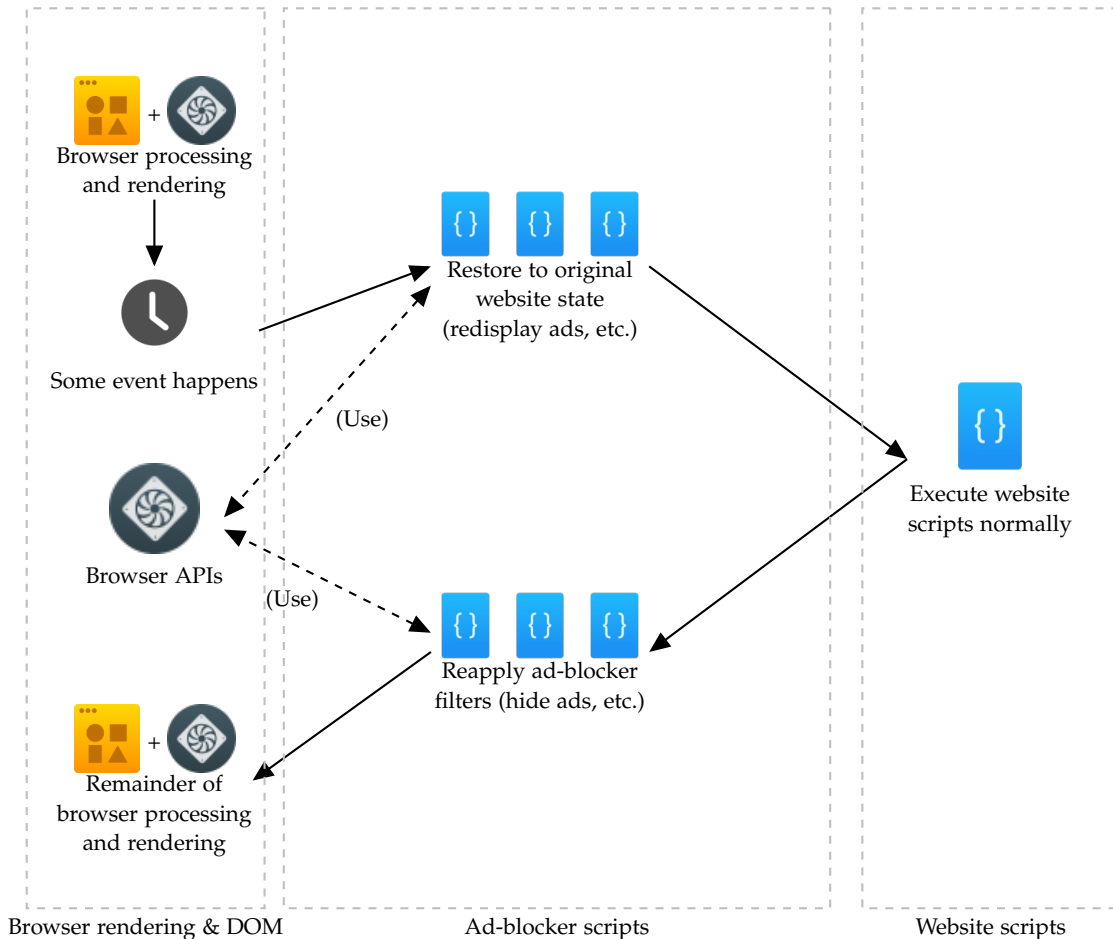


4.4.1 Website script context detection

Website script context detection is the technique that allows our prototype ad-blocker to avoid detection by ad-blocker detectors that rely on **element hiding rules**.

This diagram explains how **website script context detection** works:

Figure 24: Website script context detection



The objective of website script context detection is to **offer a different view of the DOM** to the ad-blocker and to the website. While the ad-blocker will have access to the "real" DOM (the one that is seen by the user, that is, a version with all the filters applied), the website scripts will have access to a version of the DOM that looks like the website with no filters applied.

Since the browser only offers a single DOM to be displayed to the user, a way to achieve this is to temporarily disable the filters of the ad-blocker, let the website run its scripts (which may include ad-blocker detectors, which will not be able to detect any unexpected

changes to the DOM since it has been restored to its original state), then reapply the filters again.

It should be noted that restoring the DOM to its original state, then reapplying the filters, does *not* necessarily cause any visual change to the user, such as a short flicker of the website's advertisements. Browsers only render the website at certain points, so if done with the right sequencing, this effect can be avoided.

Being able to restore the original website state is not technically challenging, though it requires some bookkeeping from the ad-blocker engine. The ad-blocker must save all elements that have been filtered, along with the original state of all properties that are changed by the filter (e.g. the value of the CSS `display` style if `display:none` is being applied), in order to be able to restore them.

On the other hand, being able to run the ad-blocker **right before** any website script executes (to restore the original website's DOM) and **right after** any website script executes (to restore the filtered DOM) is more challenging. We tested two approaches: **Modifying the original website scripts** to insert callbacks into the ad-blocker, or **hooking DOM access APIs** to receive callbacks before any DOM access.

Figure 25: Example ad-blocker detector code

```
1 <script>
2 window.onload = function() {
3     setTimeout(function() {
4         var baitElement = document.getElementById("baitElement");
5         if (baitElement.clientHeight === 0)
6             alert("Ad-Blocker Detected");
7         baitElement.parentNode.removeChild(baitElement);
8     }, 500);
9 };
10 </script>
11 <div class="adsbox" id='baitElement'>BAIT</div>
```

The original way we used relied on **modifying the original website scripts**. We used a script parsing library called *acorn* [49] in order to parse the website's scripts and insert callbacks into our ad-blocker code before and after any function call:

Figure 26: Ad-blocker detector code, modified by script rewriting to call the ad-blocker to restore/reapply the original website state

```
1 <script>
2 +window.adBlocker.restoreWebsiteState();
3 window.onload = function() {
4 +   window.adBlocker.restoreWebsiteState();
5     setTimeout(function() {
6 +       window.adBlocker.restoreWebsiteState();
7         var baitElement = document.getElementById("baitElement");
8         if (baitElement.clientHeight === 0)
9             alert("Ad-Blocker Detected");
10        baitElement.parentNode.removeChild(baitElement);
11 +       window.adBlocker.reapplyWebsiteState();
12    }, 500);
13 +   window.adBlocker.reapplyWebsiteState();
14 };
15 +window.adBlocker.reapplyWebsiteState();
16 </script>
17 <div class="adsbox" id='baitElement'>BAIT</div>
```

The other technique an adaptation from the techniques of *The Future of Ad Blocking* [28]. In order to detect when a website script starts executing (actually, when it first accesses a DOM element), we hook the DOM element functions in order to receive a callback when any of them is called. This way, we can receive a callback before the access happens and trigger the DOM.

With this technique, we can receive a callback *before* the website accesses a DOM element, but doesn't provide us with any way to receive a callback *after* the website script finishes executing. However, certain browser processes can be scheduled to be delayed between the script execution and the render process, by using the "microtask stack" [30]. There are various ways to schedule tasks to the microtask task, such as resolving a *Promise*. This offers us a window of opportunity to reapply the filters to the DOM after the website's scripts have finished executing, but without causing any visible effect to the user.

Figure 27: Ad-blocker detector code, along with sample code intercepting DOM APIs in order to restore and reapply the original website state

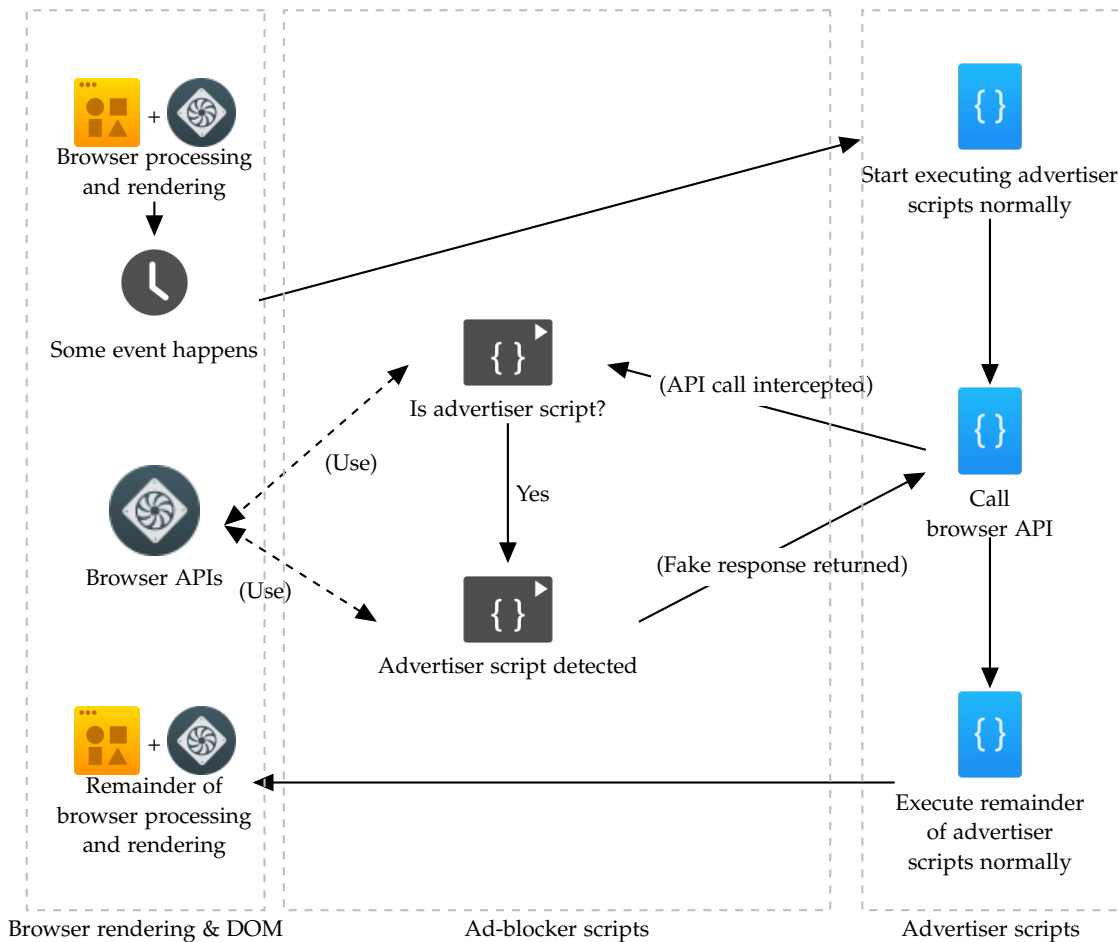
```
1 <script>
2 var desc = Object.getOwnPropertyDescriptor(
3     Element.prototype, 'clientHeight');
4 Object.defineProperty(Element.prototype, 'clientHeight', {
5     get: function() {
6         window.adBlocker.restoreWebsiteState();
7         Promise.resolve().then(window.adBlocker.reapplyWebsiteState);
8         return desc.get.call(this);
9     },
10    set: function(newValue) {
11        window.adBlocker.restoreWebsiteState();
12        Promise.resolve().then(window.adBlocker.reapplyWebsiteState);
13        desc.set.call(this, newValue);
14    },
15    enumerable: desc.enumerable,
16    configurable: desc.configurable
17 });
18
19 window.onload = function() {
20     setTimeout(function() {
21         var baitElement = document.getElementById("baitElement");
22         if (baitElement.clientHeight === 0)
23             alert("Ad-Blocker Detected");
24         baitElement.parentNode.removeChild(baitElement);
25     }, 500);
26 };
27 </script>
28 <div class="adsbox" id="baitElement">BAIT</div>
```

4.4.2 Advertiser script context detection

Advertiser script context detection is the technique that allows our prototype ad-blocker to allow execution of advertiser scripts, that is, scripts that are filtered according to the current filter lists (which is required in order to avoid detection by ad-blocker detectors that rely on **address blocking rules**), while at the same time being able to apply filters to the advertisements created by the advertiser scripts, and mitigate the privacy consequences of their execution.

This diagram explains how **advertiser script context detection** works:

Figure 28: Advertiser script context detection



The objective of advertiser script context detection is to **run advertiser scripts** in a 'virtualized' environment that can be effectively controlled by the ad-blocker.

In this virtualized environment, we want to receive callbacks from two kinds of browser actions that we want to control: The first is creation or modification of elements by the advertiser scripts (e.g. by using `document.createElement(...)`, `document.write(...)`, `element.innerHTML = "..."`, etc., since those scripts may create elements that we would otherwise have avoided by blocking the entire script, and the second is to control access to privacy-sensitive browser APIs (e.g. location), in order to mitigate the privacy consequences of executing those scripts. We also need to be able to distinguish whether those actions were done by a regular website script, or an advertiser script, since we will receive callbacks in either case by using the APIs provided by the browsers.

In order to **distinguish whether those actions were done by a regular website script,**

or an advertiser script, once we receive a callback, we tested two techniques. The first one is to **modify the original website scripts**, similarly to what we did for website script context detection. The second one is to use the non-standard `Error.prototype.stack` property [35] to check the call stack in order to detect if the current invocation of the API originated in an advertiser script or not.

It should be noted that while this API is not standardized, it is implemented (in slightly different ways) in most mainstream browsers, such as *Mozilla Firefox*, *Google Chrome*, and *Microsoft Edge*. We used the *stacktrace-js* [14] library in order to simplify access to the call stack and abstract away browser differences. From that point on, we check if any of the functions in the current call stack are defined in one of the filtered advertiser scripts and react accordingly.

In order to **receive callbacks when a DOM element is created or modified**, we can use the standard *MutationObserver* [35] browser API. However, due to the way this API works (execution is delayed until the script execution ends), it will not allow us to distinguish the origin of the action using the techniques we explained previously. However, a now deprecated but still widely supported API called *mutation events* [35] allows us to receive a callback immediately after the DOM element creation or modification, thus being able to detect which elements have been created by the advertiser script, therefore being able to block them and making previously explained techniques applicable as well.

In order to **control access to privacy-sensitive browser APIs**, we hook the functions defined in the various privacy-sensitive browser APIs such as `XMLHttpRequest`, `MouseEvent`, `navigator.geolocation`, etc., similarly to how we did for **intercepting DOM APIs** in website script context detection, and if we detect they originate in an advertiser script, we return a valid but falsified response for them that avoids revealing potentially private user data.

4.5 Implementation difficulties

In this section, we examine some of the issues that we found while developing the prototype and the solutions we have adopted or propose for those.

4.5.1 IFrames and domain restrictions

Through the `<iframe>` element, it is possible to embed a different document or website into an area of a website. The most common use case of `iframes` are to embed videos or advertisements from third-party providers.

Figure 29: YouTube video embedded in an external website. Source: *EngineerGuy.com*



An `iframe` defines a completely new and independent DOM tree, but depending on the circumstances (such as the same-origin policy [35] and the sandboxing options [35]), the embedded `iframe` is able to interact with the website that defines the `iframe` or not.

While this doesn't pose any fundamental problem for implementing the techniques we propose, it adds notable implementation difficulty, since an ad-blocker detector could operate among various different `iframes`, so an approach that can operate for each group of `iframes` that can interact with each other is needed.

Our approach to this problem has been to initialize a single instance of our ad-blocker prototype for each group of `iframes` that can interact with each other, which manages the entire set of DOM trees corresponding to the `iframes` as a whole, thus being able to deal with ad-blocker detectors that operate among various different `iframes`. Other imple-

mentation approaches are possible, though overall the problem adds significant complexity to the implementation due to the variety of cases that should be handled (cross-`iframe` interactions, an `iframe` being reloaded, etc.).

4.5.2 Various modes to load scripts, images, etc.

Similarly to the previous issues, there are also various ways to load the scripts and images (for example, scripts can be executed with a regular `<script> ... </script>` element, with a `window.eval(...)` call, among others, and images can be loaded by an `` element, a `background-image: url(...)` CSS style, among others, etc.

For our prototype, we have not tried to exhaustively cover all modes to load scripts, images, and other resources, but we have sampled a notable selection of the various ways to load those resources in order to examine and solve the various difficulties that could occur.

4.5.3 Content-types and how browsers detect content

In order to inject our ad-blocker (or modify the scripts for website script context detection or advertiser script context detection), when we receive a request in our man-in-the-middle proxy, it's necessary to detect which type of file is being requested: An HTML page, a JavaScript script, an image, a movie, etc..

Real-world web browsers 'guess' which type of file is being requested based on its contents; if a file looks like something that is HTML, such as if it starts with a sequence such as `<html><head><title>My Web...`, then the web browser will recognize the file as an HTML document and display it accordingly.

However, each browser has slightly different rules for guessing the type of the file, which can cause some difficulties around edge cases such as malformed or ambiguous documents.

A simpler way to detect the type of the document is to use the MIME type of the file, which is provided by the website servers in the request response `Content-Type` header. For example, an HTML document has a MIME type of `text/html`.

While this method is not perfect (there are often multiple MIME types for the same file type, and some servers will send invalid MIME types, since browsers generally ignore this information and thus it doesn't cause any visible effect), we used this method in the prototype due to it being simpler than the other methods and being much easier to implement.

4.5.4 Performance

When a website is loaded under our prototype, it generally has a higher loading time than the time that would be spent loading normally. While this performance penalty is not severe, it is noticeable during normal use. We have found various factors that contribute

to this performance penalty, some of which are due to implementation details of our prototype, and some of which are due to the techniques used:

- The usage of a man-in-the-middle proxy adds some overhead to every HTTP request made by the browser.

This performance bottleneck **can be mitigated by implementing our ad-blocker as a browser add-on.**

- Unlike an ad-blocker that works like a browser add-on, our prototype ad-blocker has to be completely reloaded for each different website.

This performance bottleneck **can be mitigated by implementing our ad-blocker as a browser add-on.**

- Ad-blockers that work as browser add-ons generally use a facility provided by the browser exclusively to add-ons, in order to hide advertisements, which is the ability to define *user-defined style sheets*, which are style sheets that are defined in a higher level than the website style sheets and therefore cannot be override by the websites.

Due to both the fact that our ad-blocker doesn't work as a browser add-on, and the techniques we implemented, we cannot use this facility. Instead, we use a regular style sheet and use the *MutationObserver* browser API to detect and block advertisements. The usage of this API has slightly worse performance characteristics than *user-defined style sheets*.

This performance bottleneck **cannot be mitigated, but should be small.**

- Additionally, the usage of *MutationObserver* currently has a severe limitation for our use case, which is the inability to recover the call stack of the code that inserted the DOM node, which is required for **advertiser script context detection** (due to *MutationObservers* working asynchronously [30]). To work around this, we are using the legacy *mutation events* browser API, which is known to have even worse performance than the *MutationObserver* browser API [35].

As far as we know, this performance bottleneck **cannot be mitigated with the current browser APIs, but is technically possible to mitigate by future browser APIs.**

- Due to 'hooking' various browser APIs for website script context detection / advertiser script context detection, there is some overhead in every usage of those properties.

This performance bottleneck **cannot be mitigated, but should be small.**

- Due to the way **website script context detection** works as proposed (by displaying all the advertisements, executing the script normally, and hiding all the advertisements again), there is a big performance impact from the browser requiring to recalculate the positions and sizes of the various elements on the DOM tree (known as reflow or layout thrashing [43]).

While in our prototype we haven't tested or implemented techniques to work around this problem, all of the following could be used to mitigate this problem:

- One could use the technique described in *The Future of Ad Blocking* [28] to cover the advertisements instead of hiding them. Similarly, one can use a CSS style that doesn't change most layout properties (`visibility:hidden` vs `display:none` [57]) to hide the advertisements.
- The cost of reflow or layout trashing can be avoided or mitigated by techniques such as caching known values, selectively restoring only the current or all advertisements depending on the accessed values, etc.

4.5.5 "Catch-all" solution to performance problems

Due some of the performance bottlenecks described being unavoidable when using the techniques proposed, we propose a **simple "catch-all" technique that can be used to improve overall performance** to levels similar of using one of the contemporary ad-blockers (which, perhaps surprisingly, tend to be better than the overall performance levels of *not* using an ad-blocker).

By default, an ad-blocker would not apply the proposed techniques, and therefore would work like the contemporary ad-blockers. However, if such a filter option were present, the anti-ad-blocker detection techniques would be applied. This would be controlled by an additional filter options could be added in the filter lists that specifies if the proposed techniques for avoiding ad-blocker detectors should be used or not.

This would be similar in implementation to the existing *generichide* and *genericblock* filter options already implemented in Adblock Plus filter lists, which were also designed with the aim of avoiding detection by ad-blocker detectors [19].

Figure 30: **Example of the currently existing *generichide* and *genericblock* filter options (excerpt from *EasyList* [11]). Those options disable generic filters (often employed by ad-blocker detectors to detect ad-blockers) for the entirety of the specified websites.**

```

1  ...
2  @@||oload.tv^$genericblock, generichide
3  @@||openload.co^$genericblock, generichide
4  ...

```

It should be noted that such filter option **does not add the same costs** that are incurred by other existing anti-ad-blocker detector solutions such as *Anti-AdBlock Killer* [47]. In those solutions, **specific workarounds for ad-blocker detectors are created for each individual site**, which generally require (a) a technical person finding and examining the ad-blocker detector, (b) a more or less complex workaround being implemented, and (c) ongoing maintenance of the workaround as the website changes. The proposed filter option could be controlled by a non-technical filter list maintainer or even the end user of the ad-blocker, and would not require any website-specific workaround or maintenance.

5 Results

We evaluated our ad-blocker prototype from three sources: An internally developed set of **test cases**, example test websites provided by **websites that offer ready-made ad-blocker detectors**, and a set of **real websites containing ad-blocker detectors** that we obtained from a crowd-sourced list.

We set up three environments for our test:

- **No ad-blocker environment:** An installation of *Google Chrome* on *Linux* with no ad-blocker extensions.
- **Regular ad-blocker environment:** An installation of *Google Chrome* on *Linux* with *AdBlock Plus* [16] configured with the *EasyList* [11] filter list.
- **Ad-blocker prototype environment:** An installation of *Google Chrome* on *Linux* configured to use our ad-blocker prototype.

During development of our ad-blocker prototype, we have developed in parallel a **set of synthetic test cases** (see 8.1.7)) designed to exercise our ad-blocker detector prototype, which we distribute along with its source code. We have checked that we avoid detection by all ad-blocker detection test cases in the *no ad-blocker environment* and *ad-blocker prototype environment*, while all tests fail in the *regular ad-blocker environment*.

Additionally, we have randomly selected **3 websites that offer ready-made ad-blocker detectors** by doing a *Google* search. We checked that those ad-blocker detectors returned a "not detected" result for the *no ad-blocker environment* and a "detected" result for the *regular ad-blocker environment*. On the *ad-blocker prototype environment*, we get a "not detected" result in 2 out of the 3 ad-blocker detectors. A look at the failing case revealed that the website was incorrectly sending `Content-Type` headers which avoided our ad-blocker from functioning correctly (a known limitation not related to the proposed techniques; see 4.5.3). An ad-hoc fix for this limitation managed to avoid detection.

Finally, we have randomly selected **8 real websites containing ad-blocker detectors** from the *Anti-AdBlock Killer* [47] list of supported websites, confirming that those websites showed an observable message for the *regular ad-blocker environment* that was absent in the *no ad-blocker environment*. On the *ad-blocker prototype environment*, we successfully avoided detection by all 8 real websites, with no advertisements being shown on the website. There was a notable performance impact (see 4.5.4) in 3 out of the 8 websites, though overall the load times remained reasonable.

6 Limitations

6.1 Browser support

We have tested our prototype with the *Google Chrome* and *Mozilla Firefox* web browsers, which are the most popular open-source browsers and the browsers that currently have the highest degree of support for the HTML5 standard.

Almost all the code for our prototype follows the currently defined HTML, CSS and JavaScript standards. However, some of the features we have used such as CSS *variables* are currently not supported by all mainstream browsers, despite being already standardized.

Additionally, some notable exceptions are the usage of deprecated *mutation events* [35] and the non-standard `Error.prototype.stack` property [35], which could cause problems with support for other browsers or new versions of the currently supported browsers.

However, all of those problems can be worked around with some implementation effort, so our techniques remain portable to other modern browsers.

6.2 Detection of the prototype through its 'footprint'

While our ad-blocker prototype makes ad-blocker detectors unable to detect our ad-blocker through the current techniques (element hiding rules or address blocking rules), future ad-blocker detectors may try to detect our prototype through the 'footprint' that our ad-blocker detector leaves: Overwritten browser APIs, additional scripts inserted into the page, etc.. However, all of those problems were solved in *The Future of Ad-Blocking* [28]:

- The fact that **native browser APIs have been overwritten** can be detected by the website by using the `Function.prototype.toString` function. However, using the 'rootkit-style' techniques in *The Future of Ad-Blocking*, this function can also be overwritten, hiding both the fact that browser APIs have been overwritten and that this same function has been overwritten, providing an 'airtight' system that prevents websites from detecting the ad-blocker [28].
- The fact that we **inject extra scripts and style sheets into the page** (and their associated DOM elements) can be detected by the website using the browser APIs that allow the website to walk the DOM. One can prevent this in two ways:

The simplest is implementing the ad-blocker as a browser extension, which allows it to work in a separate environment as the rest of the website. In this way, making the ad-blocker resources inaccessible to the website is already handled by the existing browser APIs.

An alternative way it to randomize their names when they are injected (so they can't be accessed directly by their class/variable names), and also make them impossible to enumerate by overwriting further browser APIs, which can be hidden using the previous 'rootkit-style' techniques.

Therefore, the current "temporary" step ahead over ad-blocker detectors of our prototype can be made into a **permanent** step ahead over ad-blocker detectors using those techniques.

6.3 Visual tricks for ad-blocker users

There are other techniques that can be used to display or hide part of the website content for ad-blocker users, albeit they are only visual tricks and not proper ad-blocker detector scripts.

Those techniques are not in the scope of this thesis and can be already dealt with with changes on filter lists (either by adding more filters, or using filter options such as *generichide* or *genericblock* [19]).

Figure 31: Example visual trick for ad-blocker users. Since the CSS class 'adsbox' is included in the popular EasyList filter list, ad-blocker users see the first layer only, while non ad-blocker users see both layers, with the second layer overlaying the first.

```
1 <style>
2 .layer
3 {
4     position:absolute;
5     top:0; left:0;
6     width:250px; height:100px;
7     border: 5px solid red; background-color: white;
8 }
9 </style>
10 <div class='layer'>You are using an ad-blocker.</div>
11 <div class='layer adsbox'>You are not using an ad-blocker.</div>
```

Figure 32: Result without a filter list-based ad-blocker

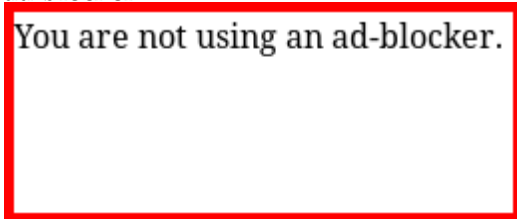


Figure 33: Result with a filter list-based ad-blocker



7 Conclusions and future directions

In this thesis, we have examined the current situation of online advertisement blocking and studied the impact of ad-blocker detectors, in terms of the motivations for publishers and advertisers to use them. We also studied the current techniques and software that are currently available to avoid detection by such blockers, and the motivation that users have to use them. Furthermore, we have proposed improvements to those techniques that allows them to work in a more generic way than the current techniques, implemented them in a prototype ad-blocker and evaluated the results of using such techniques.

Such techniques are currently not implemented into any of the fully-featured mainstream browser extensions such as *AdBlock Plus* and *uBlock Origin*, but we expect that they can be implemented if there is enough interest from those browser extensions. However, they need to be polished and their limitations should be addressed to the maximum extent in order to be applicable to regular browsing.

Even though *The Future of Ad Blocking* [28] provides novel ad-blocking techniques and concludes that ultimately ad-blockers can be made undetectable to ad-blocker detectors, we don't expect the so-called "ad-blocking war" to be over anytime soon, since publishers and advertisers can easily implement new ways to serve advertisements that can better survive contemporary ad-blockers. Some examples of this could be serving advertisements from first-parties in an obfuscated way, or embedding advertisements in media content using the recent HTML5 encrypted media extensions. Further work can examine how such changes could be implemented and their consequences for the Web.

In addition, further work in novel ad-blocking and anti-tracking techniques using heuristics is also a potentially fertile land for new research to be done, with new techniques being currently contributed by both academia, pro-privacy organizations and industry leaders alike.

The current landscape of interactions between users, publishers, advertisers and tracking third-parties is a complex issue in constant tension and with non-trivial consequences for the future of the Web, and everyone should watch with interest how such relationships evolve in order to find a way to make the Web sustainable while avoiding current threats to users.

8 Appendices

8.1 Testing the prototype

Here we expose some brief instructions to get the prototype ad-blocker we have developed implementing the proposed techniques running.

8.1.1 Installing *Node.js*

You will need to install *Node.js* [39] in order to run the prototype. We tested it under various recent versions, though we recommend using *v6.11.0 LTS*.

8.1.2 Obtaining the prototype

You should have received a copy of the prototype along with this document. Otherwise, you can download it from <https://bitbucket.org/joanbrugueram/content-filter-policy> using *Git*.

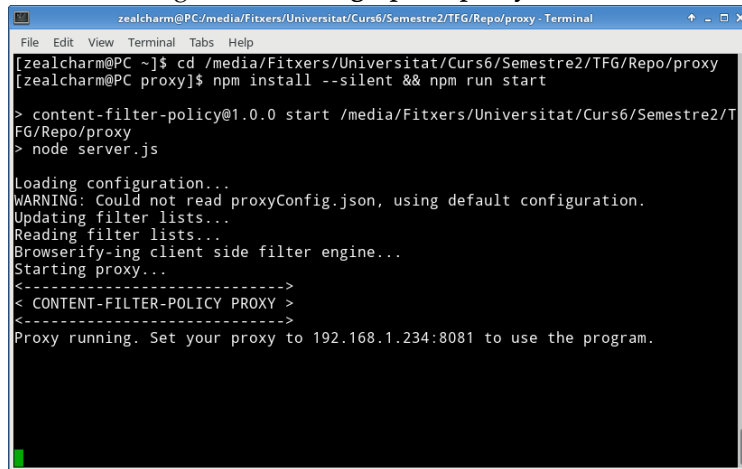
Inside the copy of the prototype, there should be three folders:

- **proxy:** The implementation of the proxy server. This is the core of the ad-blocker prototype.
- **test_cases:** This is a set of tests that can be used to test the effectiveness of the prototype in a consistent way.
- **thesis:** This is the \LaTeX source of this document.

8.1.3 Installing the dependencies and starting the proxy server

In order to install all the required dependencies and start the proxy server, you should open a terminal window and type `npm install && npm run start`. This will start the proxy server at your current local IP address on port 8081. You should keep this terminal window open during the next steps.

Figure 34: Setting up the proxy server



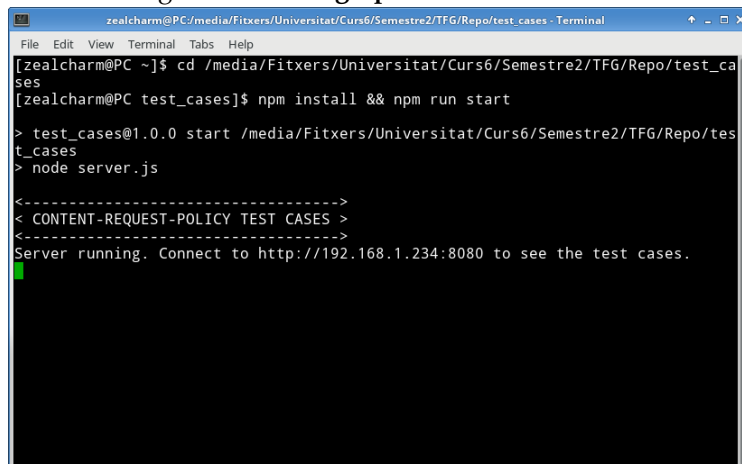
```
zealcharm@PC:/media/Fitxers/Universitat/Curs6/Semestre2/TFG/Repo/proxy - Terminal
File Edit View Terminal Tabs Help
[zealcharm@PC ~]$ cd /media/Fitxers/Universitat/Curs6/Semestre2/TFG/Repo/proxy
[zealcharm@PC proxy]$ npm install --silent && npm run start
> content-filter-policy@1.0.0 start /media/Fitxers/Universitat/Curs6/Semestre2/TFG/Repo/proxy
> node server.js

Loading configuration...
WARNING: Could not read proxyConfig.json, using default configuration.
Updating filter lists...
Reading filter lists...
Browserify-ing client side filter engine...
Starting proxy...
<----->
< CONTENT-FILTER-POLICY PROXY >
<----->
Proxy running. Set your proxy to 192.168.1.234:8081 to use the program.
```

8.1.4 Starting the test case server

In order to install all the required dependencies and start the test case server, you should open a terminal window and type `npm install && npm run start`. This will start the test case server at your current local IP address on port 8080. You should keep this terminal window open during the next steps.

Figure 35: Setting up the test case server



```
zealcharm@PC:/media/Fitxers/Universitat/Curs6/Semestre2/TFG/Repo/test_cases - Terminal
File Edit View Terminal Tabs Help
[zealcharm@PC ~]$ cd /media/Fitxers/Universitat/Curs6/Semestre2/TFG/Repo/test_cases
[zealcharm@PC test_cases]$ npm install && npm run start
> test_cases@1.0.0 start /media/Fitxers/Universitat/Curs6/Semestre2/TFG/Repo/test_cases
> node server.js

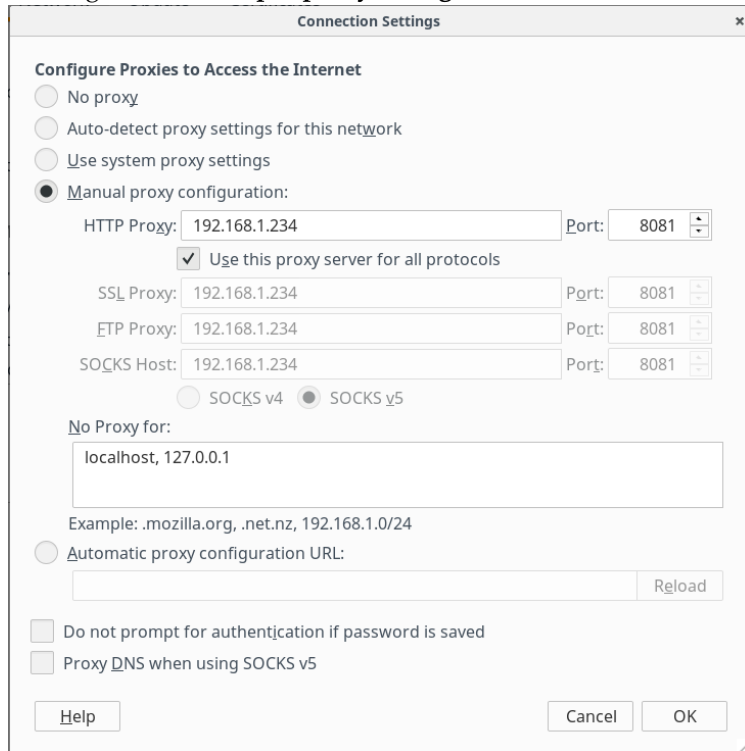
<----->
< CONTENT-REQUEST-POLICY TEST CASES >
<----->
Server running. Connect to http://192.168.1.234:8080 to see the test cases.
```

8.1.5 Browse the test cases with a web browser

Start by completely clearing your web browser's cache and disabling any currently installed ad-blockers, since those may infer with the operation of the prototype.

You should then configure your web browser or operating system to connect through the proxy server at your IP address and port 8081. This will depend on the web browser or operating system.

Figure 36: Example proxy configuration with Firefox



After this, opening the URL at your IP address and port 8080 (`http://YOUR.IP.ADDRESS.HERE:8080`) will show you the list of the test cases.

Figure 37: Browsing the test cases with Firefox

content dom height dynamic temporary	content dom height dynamic temporary jquery
Adblocker Not Detected	Adblocker Not Detected
content dom height static	content dom height static eventhandler
Adblocker Not Detected	Adblocker Not Detected
content dom height static exotic css display	content dom height static externaljs
Adblocker Not Detected	Adblocker Not Detected

8.1.6 Further steps

- **Editing the configuration file:** You can fine-tune various options (filter lists, options for the techniques, etc.) by copying the sample file at `proxy/proxyConfig.json.defaults` to `proxyConfig.json` and editing it accordingly. The configuration file contains details about the meaning of each available configuration.
- **Browsing secure (HTTPS) websites with the prototype:** Due to browser security measures, browsers will not allow navigating to secure (HTTPS) websites under a man-in-the-middle proxy. If you wish to authorize your browser to browse HTTPS websites under the prototype ad-blocker, you should import the certification authority defined under the (hidden) `proxy/.http-mitm-proxy/certs/ca.pem` file to your web browser.

8.1.7 Test cases

The folder `test_cases` contains a set of tests designed to exercise ad-blockers in order to test if they are effective, and their ability to be detected.

Those test cases have been developed based on a using both ready-made ad-blocker detector solutions (such as *BlockAdBlock* [48]), practical examples analyzed by research [38] and our own knowledge of ad-blocker detector techniques.

The test cases are typically brief and self-contained, so they can be easily understood by looking in the source code.

On the folder, the test cases are classified under different prefixes, namely:

- **content_*:** Tests related to ad-blocker detection through **element hiding rules**.
- **request_*:** Tests related to ad-blocker detection through **address blocking rules**.
- **correctness_*:** Tests of the implementation of the prototype in various "tricky" cases.
- **privacy_*:** Tests of the implementation in the prototype of privacy-preserving techniques.
- **performance_*:** Performance tests of the implementation in the prototype.
- **outofproject_*:** Tests of the limitations of the prototype.

The expected results of the test cases without an ad-blocker, with a regular blacklist-based ad-blocker (such as *uBlock Origin*), and under the prototype, are:

Figure 38: Expected results of the test cases without an ad-blocker, with a regular blacklist-based ad-blocker, and with our prototype

Expected result	Without an ad-blocker	With an ad-blocker	With the prototype
<code>content_*</code> :	Ads + Ad-blocker not detected	No ads + Ad-blocker detected	No ads + Ad-blocker not detected
<code>request_*</code> :	Ads + Ad-blocker not detected	No ads + Ad-blocker detected	No ads + Ad-blocker not detected
<code>correctness_*</code> :	OK	OK	OK
<code>privacy_*</code> :	Some OK, some fail	Some OK, some fail	OK
<code>performance_*</code> :	OK	OK	Some OK, some fail
<code>outofproject_*</code> :	OK	Some OK, some fail	Some OK, some fail

8.2 Browser fingerprinting by enabled filter lists

One can use the same techniques used to detect blacklist-based ad-blockers by websites to create a fingerprint based on which filter lists the user has enabled.

The technique used to create the fingerprint is simple in the face of how ad-blocker detector lists work. Since there are relatively few widely used filter lists, which usually contain hundreds of thousands of filters, there is, with a very high degree of certainty, a set of filters that is unique to that filter list. By detecting whether or not those filters are being applied by a web browser (a specific DOM element is hidden or an address is blocked), they can be used to detect whether this filter list is enabled or not.

This fingerprinting technique can be used without requiring the user to have JavaScript enabled, since ad-blocker detection by address blocking rules can be done by the website server without relying on client-side scripts. This is remarkable, since JavaScript is the main vector used for fingerprinting, and is typically disabled by those trying to avoid being fingerprinted, such as users of *Tor* or *i2p*.

Additionally, on most cases, we expect that this fingerprint not to be enough to track individual users on his own, due to the high prevalence of users with the same ad-blocker settings.

An illustration of fingerprinting by filter lists can be seen in the `privacy_fingerprint_ad_lists` test case.

References

- [1] Add-ons for Mozilla Firefox, *Most Popular Extensions [for Mozilla Firefox]*, <https://addons.mozilla.org/en-US/firefox/extensions/?sort=users>
- [2] AdFender, *AdFender - AdFender gives you a faster, less cluttered and more private web surfing experience!*, <https://www.adfender.com>
- [3] AdGuard, *AdGuard - The world's most advanced ad blocker!*, <https://adguard.com>
- [4] Brave browser (Brave Software), *What is Ad Replacement?*, https://www.brave.com/about_ad_replacement.html
- [5] Brave Software, *Brave Software - Building a Better Web*, <https://www.brave.com/>
- [6] Brian R. Bondy et al, *abp-filter-parser (Github)*, <https://github.com/bbondy/abp-filter-parser>
- [7] Browserling, *Browserify*, <http://browserify.org/>
- [8] Chrome Web Store, *Chrome Web Store*, <https://chrome.google.com/webstore>
- [9] CNN, *"Super cookies" track you, even in privacy mode*, <http://money.cnn.com/2015/01/09/technology/security/super-cookies/index.html>
- [10] Daniel Aleksandersen, *EFF's Privacy Badger will deteriorate your browser experience*, <https://ctrl.blog/entry/privacy-badgering>
- [11] EasyList Filter List Project, *EasyList - Overview*, <https://easylist.to/>
- [12] Electronic Frontier Foundation, *Electronic Frontier Foundation - Privacy Badger*, <https://www.eff.org/privacybadger>
- [13] Electronic Frontier Foundation, *Panopticlick - What is browser fingerprinting?*, <https://panopticlick.eff.org/about#browser-fingerprinting>
- [14] Eric Wendelin, Victor Homjakov, Oliver Salzburg et al., *JavaScript framework-agnostic, micro-library for getting stack traces in all web browsers*, <https://www.stacktracejs.com/>
- [15] Eyeo GmbH, *Allowing acceptable ads in Adblock Plus*, <https://adblockplus.org/acceptable-ads>
- [16] Eyeo GmbH, *Adblock Plus - Surf the web without annoying ads*, <https://adblockplus.org/>
- [17] Eyeo GmbH, *Adblock Plus filters*, <https://adblockplus.org/en/filters>
- [18] Eyeo GmbH, *Adblock Plus user survey results*, <https://adblockplus.org/blog/adblock-plus-user-survey-results-part-1>, <https://adblockplus.org/blog/adblock-plus-user-survey-results-part-2>, <https://adblockplus.org/blog/adblock-plus-user-survey-results-part-3>

- [19] Eyeo GmbH, *New filter options \$generichide and \$genericblock*, <https://adblockplus.org/development-builds/new-filter-options-generichide-and-genericblock>
- [20] FiveFilters, *There are no acceptable ads*, <https://github.com/fivefilters/block-ads/wiki/There-are-no-acceptable-ads>
- [21] Forbes, *Inside Forbes: More Numbers On Our Ad Blocking Plan – and What’s Coming Next*, <https://www.forbes.com/sites/lewisdvorkin/2016/02/10/inside-forbes-more-numbers-on-our-ad-blocking-plan-and-whats-coming-next/>
- [22] Google, *Block or allow pop-ups in Chrome*, <https://support.google.com/chrome/answer/95472?co=GENIE.Platform%3DDesktop&hl=en>
- [23] Google, *Google Contributor - Introduction*, <https://contributor.google.com>
- [24] gorhill (Raymond Hill), *uBlock Origin - An efficient blocker add-on for various browsers. Fast, potent, and lean.*, <https://github.com/gorhill/uBlock>
- [25] gorhill (Raymond Hill), *uBlock vs. ABP: efficiency compared*, <https://github.com/gorhill/uBlock/wiki/uBlock-vs.-ABP:-efficiency-compared>
- [26] gorhill (Raymond Hill), *uMatrix: Point and click matrix to filter net requests according to source, destination and type*, <https://github.com/gorhill/uMatrix>
- [27] Grant Storey, Dillon Reisman, Jonathan Mayer, Arvind Narayanan, *Perceptual Ad Highlighter*, <https://chrome.google.com/webstore/detail/perceptual-ad-highlighter/mahgifl1leahghaapkboihnbhdplhnchp>
- [28] Grant Storey, Dillon Reisman, Jonathan Mayer, Arvind Narayanan, *The Future of Ad Blocking: An Analytical Framework and New Techniques*, <http://randomwalker.info/publications/ad-blocking-framework-techniques.pdf>
- [29] InformAction, *NoScript - JavaScript/Java/Flash blocker for a safer Firefox experience!*, <https://noscript.net/>
- [30] Jake Archibald, *Tasks, microtasks, queues and schedules*, <https://jakearchibald.com/2015/tasks-microtasks-queues-and-schedules/>
- [31] Jason Koebler (Motherboard, VICE), *Princeton’s Ad-Blocking Superweapon May Put an End to the Ad-Blocking Arms Race*, https://motherboard.vice.com/en_us/article/princetons-ad-blocking-superweapon-may-put-an-end-to-the-ad-blocking-arms-race
- [32] Joe Ferner et al, *http-mitm-proxy (Github)*, <https://github.com/joeferner/node-http-mitm-proxy>
- [33] jspenguin2017, *AdBlock Protector*, <https://github.com/jspenguin2017/AdBlockProtector>

- [34] Microsoft, *Use Do Not Track in Internet Explorer 11*, <https://support.microsoft.com/en-us/help/17288/windows-internet-explorer-11-use-do-not-track>
- [35] Mozilla Developer Network, *Web technology for developers | MDN*, <https://developer.mozilla.org/en-US/docs/Web>
- [36] Mozilla Support, *Set Adobe Flash to "click to play" on Firefox*, <https://support.mozilla.org/en-US/kb/set-adobe-flash-click-play-firefox>
- [37] Mozilla, *Tracking Protection in Private Browsing*, <https://support.mozilla.org/en-US/kb/tracking-protection-pbm>
- [38] Muhammad Haris Mughees, Zhiyun Qian, Zubair Shafiq, Karishma Dash, Pan Hui, *A First Look at Ad-block Detection – A New Arms Race on the Web*, <https://arxiv.org/pdf/1605.05841.pdf>
- [39] Node.js Foundation, *Node.js*, <https://nodejs.org>
- [40] npm, Inc., *npm*, <https://www.npmjs.com/>
- [41] PageFair & Adobe, *The cost of ad blocking*, https://downloads.pagefair.com/wp-content/uploads/2016/05/2015_report-the_cost_of_ad_blocking.pdf
- [42] PageFair, *2017 Adblock Report*, <https://pagefair.com/blog/2017/adblockreport/>
- [43] Paul Irish (et al), *What forces layout/reflow. The comprehensive list. (and related links)*, <https://gist.github.com/paulirish/5d52fb081b3570c81e3a> (and related links)
- [44] Pi-Hole, *Pi-hole™: A black hole for Internet advertisements*, <https://pi-hole.net/>
- [45] Quartz Media, *The EU is totally fine with with publishers blocking ad blockers*, <https://qz.com/882462/the-eu-is-totally-fine-with-with-publishers-blocking-ad-blockers/>
- [46] Raymond.CC, *10 Ad Blocking Extensions Tested for Best Performance*, <https://www.raymond.cc/blog/10-ad-blocking-extensions-tested-for-best-performance/view-all/>
- [47] Reek, *Anti-Adblock Killer helps you keep your Ad-Blocker active, when you visit a website and it asks you to disable. (github)*, <https://github.com/reek/anti-adblock-killer>
- [48] sitexz, *BlockAdBlock - Allows you to detect the extension AdBlock (and other)*, <https://github.com/sitexw/BlockAdBlock>
- [49] Tern JavaScript tooling software, *Acorn - A tiny, fast JavaScript parser, written completely in JavaScript (Github)*, <https://github.com/ternjs/acorn>

- [50] The Chromium Projects (Google), *Chromium Security - Technical analysis of client identification mechanisms - Browser-level fingerprints*, <https://www.chromium.org/Home/chromium-security/client-identification-mechanisms#TOC-Browser-level-fingerprints>
- [51] The Guardian, *Adblock Plus launching platform to sell 'acceptable' ads*, <https://www.theguardian.com/business/2016/sep/13/adblock-plus-launching-platform-to-sell-acceptable-ads>
- [52] The Guardian, *Adblocking: advertising 'accounts for half of data used to read articles'*, <https://www.theguardian.com/media/2016/mar/16/ad-blocking-advertising-half-of-data-used-articles>
- [53] The Guardian, *Google Contributor: can I really pay to remove ads?*, <https://www.theguardian.com/technology/2014/nov/21/google-contributor-pay-remove-ads>
- [54] The Register, *Ad-blocker blocking websites face legal peril at hands of privacy bods*, https://www.theregister.co.uk/2016/04/23/anti_ad_blockers_face_legal_challenges/
- [55] The Stack, *Sites that block adblockers seem to be suffering*, <https://thestack.com/world/2016/04/21/sites-that-block-adblockers-seem-to-be-suffering/>
- [56] uBlock Origin, *Add acceptable ads (Github issue)*, <https://github.com/chrisaljoudi/uBlock/issues/66>
- [57] W3Schools, *CSS Layout - The display Property - Hide an Element - display:none or visibility:hidden?*, https://www.w3schools.com/css/css_display_visibility.asp
- [58] Webopedia, *Webopedia: Online Tech Dictionary for IT Professionals*, <http://www.webopedia.com> (you can look for the referred term here)
- [59] WIRED, *Publishers Strike Back at a Browser That Replaces Their Ads*, <https://www.wired.com/2016/04/brave-software-publishers-respond/>
- [60] WIRED, *Why Malvertising Is Cybercriminals' Latest Sweet Spot*, <https://www.wired.com/insights/2014/11/malvertising-is-cybercriminals-latest-sweet-spot/>

Icons from the Autü Plasma 5 theme available at <https://github.com/lucasnota/Antu-U>.

All web sources checked on June 2017.

You may find amended versions at <https://bitbucket.org/joanbrugueram/content-filter-policy>.