FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Visual Detection of Anti-Adblocking Software

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

Samuel Brendler, BSc.
Registration Number 0952463

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl, Projectassistent
Assistance: Dr. techn. Markus Huber, MSc

Vienna, 31st August, 2017

Samuel Brendler	Edgar Weippl

Kurzfassung

Adblocker erfreuen sich zunehmender Beliebtheit, weswegen Webseitenbetreiber begonnen haben, mit Anti-Adblockern Adblock-Benutzern den Zugang zu ihren Webseiten zu versperren. Daraus ist ein Wettrüsten entstanden, welches nicht klar entschieden ist. In dieser Arbeit werden die technischen Möglichkeiten, welche beiden Seiten zur Verfügung stehen, analysiert. In einer Fallstudie werden die Anti-Adblock Implementierungen von drei bekannten Webseiten und Gegenreaktionen der Adblock-Community untersucht. Des Weiteren wurde ein Webcrawler zur automatischen Erkennung von Webseiten, die Anti-Adblocker einsetzen implementiert. Die Erkennung basiert auf visuellen Merkmalen von Screenshots und verwendet diese um mit Algorithmen für machinelles Lernen Modelle zu generieren. Mit diesen Modellen wird ein zufällig gewählter Datensatz von 300 Webseiten aus der Alexa Topliste analysiert.

Abstract

The number of adblock users has been on the rise throughout the last years, with the consequence that publishers have started using anti-adblocking software to deny adblock users access to their websites. This has lead to an ongoing arms race between the advertising industry and adblock users from which no winner has emerged. In this thesis the technical possibilities, which both sides have at their disposal, are analyzed. In a case study the implementations of anti-adblocking scripts and the adblocking community's reactions and countermeasures are compared. Furthermore a webcrawler for automatically detecting websites, which use anti-adblocking scripts was implemented. The detection is based on visual features of screenshots and uses them with machine learning algorithms to generate models. With those models, a random sample of 300 websites from the Alexa top 1 million websites is analyzed.

Contents

CHAPTER 1

Introduction

1.1 Motivation and problem definition

Online advertising has been a driving factor for the rapid growth of the Internet as we know it today because it enables content providers (e.g. website owners) to create revenue without directly charging their users. Over the last years the global market for online advertising has been growing continuously, reaching 145 billion USD in 2014 according to eMarketer[1] [46].

By targeting ads directly at users, advertisers can achieve better results, but also need access to private information about their users such as browsing history, context information, geolocation and social networking profiles. Users are therefore tracked across websites, which allows to create detailed profiles of them, resulting in a conflict of interests because many users consider this a cutback of their privacy. Additionally, online advertising also adds multiple other drawbacks to the user's web browsing experience. For example ads can significantly increase the loading time of websites [39], infect the user's computer with malicious code [44] (malicious advertising a.k.a. *malvertising*) and worsen the user experience by taking up space on websites or inducing unwanted waiting times when being forced to watch an ad before streaming a video.

With the increasing presence of online advertising, the usage of ad-blocking software (a.k.a. *adblockers*) such as the popular browser plugin *Adblock Plus* also became more popular. The problem with adblockers is that content providers don't profit from users, who use them. Thus, in the hypothetical scenario that the majority of users was browsing with adblockers enabled, content providers would not be able to offer their contents for free and would have to switch to other business models or be forced to shut their services down. As depicted in Figure 1.1 there is a current trend of growing adblocker user rates, which led to a still ongoing public debate about the future of online advertising.

[1] Original source not publicly accessible, therefore recited

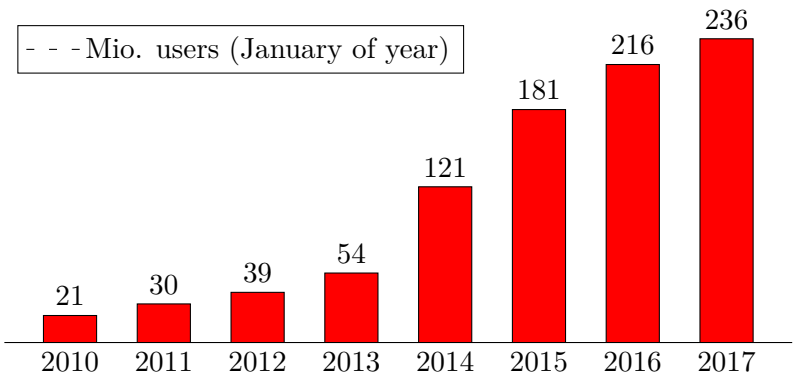

Figure 1.1: Worldwide development of desktop adblock usage
Data from Pagefair *2017 Global Adblock Report* [60]

Publishers such as print media, which are struggling to make the transition to online media in a profitable way, are trying to counteract adblocking by lobbying, lawsuits and anti-adblocking software, which detects the usage of adblockers and prohibits users with active adblockers from accessing their contents. On the opposite site there are companies like *Eyeo* (*Adblock Plus*), *Ghostery* and a large open-source community. Even though scientific research on adblockers exists, anti-adblocking software is a very recent trend, that has not yet been covered in detail. A lot of knowledge about it is distributed among blogs, software documentation and reports of companies, which provide services specialized in the field.

We have now reached a tipping point in the history of online advertising, where its future is uncertain because advertisers and adblockers have already engaged in an arms-race against each other. In many cases anti-adblocking software can already be countered by tools such as *Anti-Adblock Killer* [71].

The goal of this thesis is to understand how this conflict can further evolve, which requires understanding how far it has already escalated and which technical possibilities there are on either side for enforcing their goals. For this reason the research questions are formulated as follows:

- What are effective measures for detecting the use of adblocking tools?
- What are effective measures for circumventing adblock-detection as used in anti-adblocking tools?
- Which categories of anti-adblocking tools exist?
- How many of websites from a random sample of 100 pages from the Alexa Top 1 million websites are using anti-adblocking software and do they use hard blocking (restrict access to content) or soft blocking (hinting to disable adblocker)?

CHAPTER 2

Background

2.1 Online advertising ecosystem

2.1.1 Ad-serving

With the continuing growth of the online advertising industry, balancing the demand and supply of ads and ad space became more challenging. Whereas in the earlier days of the Internet publishers had direct contracts with advertisers or ad networks, nowadays a large share of ads are auctioned in real time to the highest bidder. The following paragraphs describe those processes based on a whitepaper by *OpenX* [57].

Ad networks serve as intermediaries between advertisers (e.g. a car company) and publishers (e.g. a news website). By contracting multiple publishers to serve its ads, an ad network creates a supply of advertising space, which it can sell to advertisers, who are in demand because they want to deliver their ads to potential consumers. The aggregation of supply and demand has multiple advantages. A publisher doesn't have to spend resources searching for advertisers who are willing to partner up with his/her website. An advertiser can increase the reach of his ads, because through the ad network, they are displayed on a large number of websites. Ad networks offer different degrees of transparency to advertisers with regards to the positioning of their ads on websites. Vertical ad networks offer high transparency and usually ad space provided by publishers from the same sector. Targeted ad networks, allow advertisers to select their target groups by defining criteria such as demographic attributes, interests and context. Blind ad networks offer

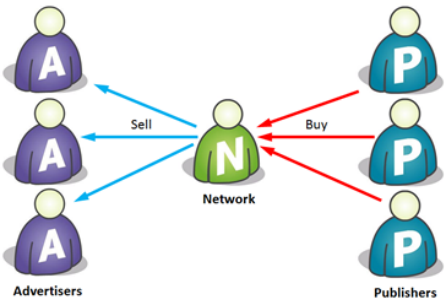

Figure 2.1: Ad networks buy ad space from publishers and sell it to advertisers

low costs for advertisers and in return take charge of the selection of publishers. A major problem of ad networks was that forecasting of their ad space supply was inaccurate. Because those spaces were sold in advance, it was possible that there were too many ads and not enough space or vice versa. Ad networks therefor started trading their capacities with other ad networks.

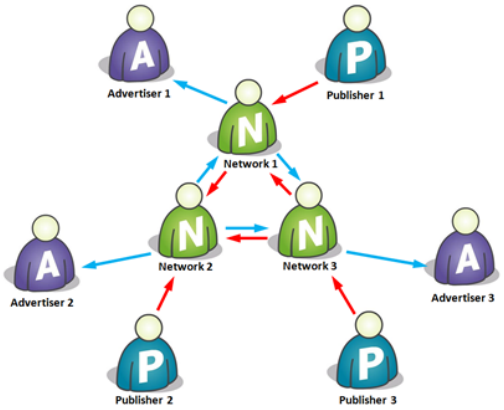

Figure 2.2: Multiple ad networks have bilateral trading relationships with each other

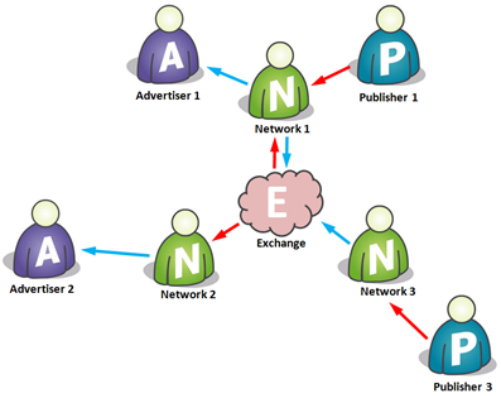

Figure 2.3: An ad exchange as single point of contact for ad networks

Source of Figures 2.1-2.3: liesdamnedlies.com [79]

Ad exchanges facilitate this process by introducing centralized marketplaces, where it is no longer necessary that ad networks negotiate individual contracts with each other. Instead publishers and advertisers have a single point of contact (per ad exchange), where ad space is auctioned to the highest bidder. The auctions are happening in real-time, which makes forecasting and planning of demand and supply unnecessary. As soon as a user visits the website of a publisher, a request is sent to the ad exchange. This request states that the publisher supplies ad space in a format specified by the website (e.g. banner ad, 800 x 200 pixels). Typically such a request also contains information about the user for allowing targeting ads. Together with context information about the originating website this user information is made available to advertisers, who can then place a bid within a time window. When the time window closes, the ad of the highest bidder gets delivered to the visitor. Because time windows are usually in the range of a few milliseconds, bidding has to happen fast and is automated with a set of bidding rules based on a bidding strategy.

Ad serving is the process of requesting and loading an ad to the browser of a website visitor. This process starts with a visitor requesting a website. The source code of the website contains *ad-tags*, which are placeholders for ads. An ad-tag contains the URL of the so called *publisher ad server* and some parameters, which identify the publisher and the characteristics of the ad space.

An example of an ad-tag of Google's *DoubleClick for Publishers* [27]:

```
http://ad.doubleclick.net/Nnetwork_code/ad/first_level_ad_unit/
second_level_ad_unit;pos=top;tile=tile_number;sz=widthxheight;
ord=1234567890?
```

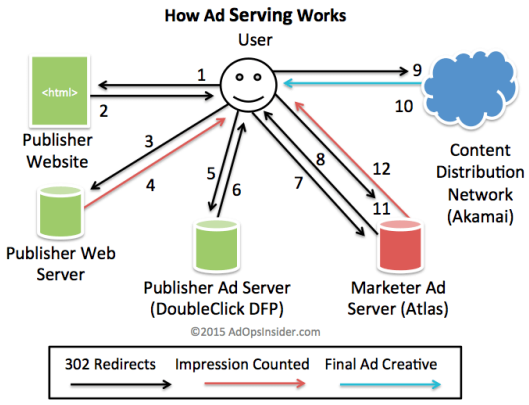

Figure 2.4: Simplified ad serving process
Image source: www.adopsinsider.com [35]

A *publisher ad server* is used for delivering and tracking advertising [35]. When the browser loads the page, it also sends a request to the publisher ad server. In a very simple scenario the publisher ad server acts as a repository for ads [80] and directly returns an ad to the user's browser. The benefits of using a publisher ad server are tracking of ad requests and separation of the website's source code and ads, which allows dynamically rotating available ads.

In a more realistic case, the publisher ad server uses external ads and redirects the browser with an HTTP 302 status code to an *advertiser ad server*. An advertiser ad server (a.k.a. *marketer ad server*) allows the advertiser to manage ad campaigns and measure delivery of ads among multiple publishers. The actual ad can be returned directly by the advertiser or by a *content distribution network*.

It is clear that performance of all involved servers is critical especially because the requests have to pass the user's browser sequentially - If there is one weak link in this chain, the overall loading time of the ad increases.

2.1.2 Adblockers

Categories of adblockers

Adblockers can be implemented in different ways. Browser extensions such as *Adblock Plus* are the first choice for most users but in some scenarios are not applicable (e.g. older iOS versions) or other solutions offer additional benefits. While researching existing solutions the following types of adblockers were identified:

Browser extensions can be installed from extension repositories, which all popular browsers offer. *Adblock Plus (ABP)*, *uBock Origin* and *Adblock* (unrelated to *ABP*) are the most prominent examples. All three of them are open-source software and rely on community-generated third party filter lists such as *EasyList*[17] or *Fanboy Annoyances List* [20]. They differ in the way they handle those filter lists internally, which leads to differences in performance and system load [47]. Because extensions have access to the browser's API, they can operate with more detailed information such as the *Document Object Model* (DOM) [42] of a website. This allows them not only to perform URL-based blocking but also to manipulate CSS values and thereby change what actually gets displayed in the visible area of the screen. Listing 2.1 shows ways to render CSS elements invisible on the screen.

```
1  /* Remove the element, occupied space collapses */
2  #blockMe1 {
3  display :none
4  }
5  /* Hide element, occupied space remains occupied */
6  #blockme2 {
7  visibility: hidden;
8  }
9  /* Resize to 0px and hide contents, occupied space collapses */
10 #blockme3 {
11 height: 0;
12 width: 0;
13 overflow: hidden;
14 }
15 /* Shift element out of visible area, occupied space collapses. Can cause
       problems with focusing links */
16 #blockme4 {
17 position: absolute;
18 top: -1000px;
19 left: -1000px;
20 }
21 /* Shift element out of visible area, occupied space remains occupied */
22 #blockme5 {
23 text-indent: -1000px;
24 }
25 /* Adding !important to the CSS property can make the rule prone to
       overwriting. */
```

Listing 2.1: CSS properties for hiding elements

Hosts file blacklisting also takes place on the user's device but filters all outbound traffic of a system according to a blacklist. The `hosts` file serves the purpose of resolving domains locally and therefor entries in it take precedence over queries to the remote DNS server. Ads can be blocked by dropping requests to ad servers and CDNs. To block requests domains are resolved to 0.0.0.0 or 127.0.0.1 instead of their real IP-addresses, which prevents them from reaching the original ad servers and CDNs. Listing 2.2 shows that traffic can be blocked for entire domains and subdomains. Even though this setup is very simplistic, it has two downsides. Firstly rules can only be defined on a domain level, which is more coarsely grained in comparison to browser extension solutions. Consequently it becomes easier for advertisers to detect that an ad has not been loaded, when a strict set of rules is active. Secondly the host file can also be abused to redirect to malicious IP addresses, which can be a problem when subscribing to a third party blacklist which is automatically updated without verifying it's contents. Currently there are multiple solutions available as open source software, some of which also block host entries of malware and tracking domains.

```
1 # Drop all traffic to exampleadserver.com
2 0.0.0.0 exampleadserver.com
3
4 # Drop all traffic to subdomain "ads" on exampleadserver.com
5 0.0.0.0 ads.exampleadserver.com
```

Listing 2.2: Example hosts file for dropping requests to ad servers

DNS-filtering is based on the same principle as hosts file blacklisting. The difference is that a typically remote DNS server takes over the task of filtering requests instead of the local machine. The advantage is that many devices can use the same DNS-server and thus the blacklist only needs to be maintained in one place. Furthermore DNS-filtering can also be applied without admin or root privileges because DNS configurations are part of standard network settings in many operating systems. This makes it a viable option on mobile devices and smart TVs, where other methods require more permissions. Even though projects for hosting an own DNS-server such as Pi-hole [62] exist, they require complex configuration and administration. Opposed to that third-party solutions such as OpenDNS [56] are easier to use because the user only has to change DNS IP-address in the network settings once and the DNS service provider handles maintenance of the blacklist. On the downside whitelisting individual websites or ad servers is not possible anymore, which can raise the issue of not being able to visit websites running anti-adblocking software. The risk of malicious redirects is the same as with hosts file blacklisting with the difference that it usually is not possible to manually inspect the blacklist. Furthermore third party DNS-services can log all DNS-requests and share their insight into the user's browsing history with marketing companies [43] and governments [29].

VPN and **proxy solutions** forward network traffic to a remote server, which takes care of the filtering. A VPN solution can apply blacklists to all the network traffic

of a computer whereas proxy solutions can be enabled per application. By providing additional encryption, VPN services can prevent man-in-the-middle attacks such as the known incidents of ad-injection by ISPs [3, 5] but on the other hand put the VPN service provider in a position to sniff and manipulate unencrypted traffic. By controlling the exit node to the Internet, the VPN service provider is in the perfect position to perform active and passive man-in-the-middle attacks. As a result the user has to entrust the provider with his data streams. When considering that many of those services are provided free of charge by for-profit companies, there is plausible reason for suspicion. Ad-free VPN service providers typically use proprietary ad-blocking technologies, which are not publicly disclosed.

Network-level adblocking is a very recent trend, where the ISP implements the filters somewhere within its infrastructure. The important difference with network-level adblocking is that users no more have to opt-in, install or configure anything - ads are blocked by default. Should network-level adblocking become prevalent among ISPs, this could drastically increase adblock user rates and seriously decrease publishers' revenues. At the time of writing, the legality of network-based adblocking is unclear. The mobile operator *Three* announced to introduce the a proprietary adblocking solution by *Shine Technologies* to all their U.K. customers [48]. Within EU borders however the *European Net Neutrality Guidelines* [54] launched in August 2016 forbid ISPs to block Internet traffic if not necessary.

Adblockers can be categorized by their scope.

- **Application-wide**
  Browser extensions
  Proxy
- **System-wide**
  Host file
  DNS
  Proxy
- **Network-wide**
  VPN
  DNS
  Gateway-/Router-based
  Proxy

Commercialization of adblocking

Adblockers can be provided as commercial products run by for-profit organizations or as non-commercial software. Commercial in this context does not necessarily imply that the user pays money to use the adblocker.

Adblock Plus is a browser extension with a very large user base provided by *Eyeo GmbH* for all major browsers. There also is a standalone Android app for system-wide adblocking. In the last years there has been a large public debate followed by lawsuits because publishers are complaining about ABP's *Acceptable Ads* program. ABP blocks ads, but whitelists ads, which are considered "acceptable". If a publisher wants his/her website to be whitelisted, it needs to comply with a list of criteria [25], apply at Eyeo and sign an agreement. Furthermore Eyeo used to charge publishers with more than 10 million ad impressions by whitelisting 30% of the additional revenue. A German court ruled in June 2016 that it is illegal to charge publishers for whitelisting. In response Eyeo now provides the acceptable ads program free of charge but with the same criteria as before [40]. Eyeo has announced a partnership with the micropayment service Flattr, which allows users to spread small amounts of money among content providers according to their browsing habits [64].

Brave [34] is a browser, which is supposed to tackle the problem of annoying ads. The basic idea is that Brave comes with an adblocker and tracking protection out-of-the-box. When surfing with Brave, intrusive ads will be blocked automatically and instead ads provided by Brave will be inserted. Revenues will be split not only between Brave and the content provider but also the user. Similar to Flattr it is also possible to transfer money to a wallet and make micropayments to content providers. Brave promises that their ads respect the users privacy and minimize page loading times. The implementation of Brave is based on the Chromium browser and still under active development, which means that many of the features are not working in the currently available build (version 0.18.23).

Subscription services Adblocking is often included in Proxy and VPN services where it helps reducing network traffic. This is especially useful with slow connections. Many of those services also include other filters such as malware domains and parental control - adblocking is just one feature among others.

Paid apps for mobile devices have received much attention lately, especially since *Apple* allowed content-blocking extensions for *iOS* 9. Usually such apps can be purchased with a one-time payment.

2.2 Problems with online ads

2.2.1 Annoying ads

From a user's perspective the most noticeable impact of online advertising is the disruption of the user experience when browsing the web. Banner and skyscraper ads often cover large areas of a page, which otherwise could be used for content (see Figure 2.5).

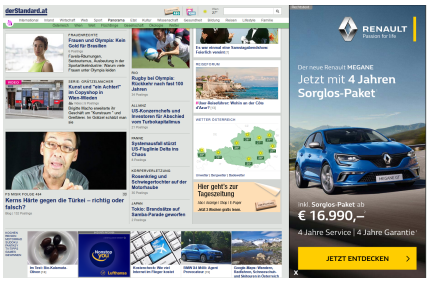

Figure 2.5: Skyscraper ad
Source: derstandard.at

Overlay ads even cover content and have to be closed by clicking a button, which often is only possible after waiting a few seconds (see Figure 2.6).

In some cases sponsored content gets embedded with a similar style as the real content , which makes it hard to tell it apart from the real content (e.g. an ad looks like an entry in a news feed - see Figure 2.7).

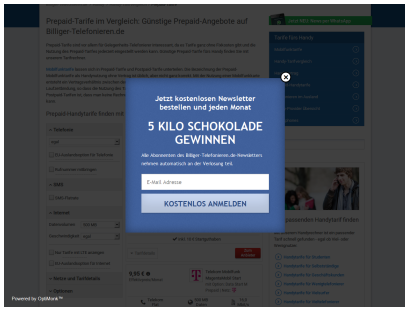

Figure 2.6: Floating Ad
Source: billiger-telefonieren.de

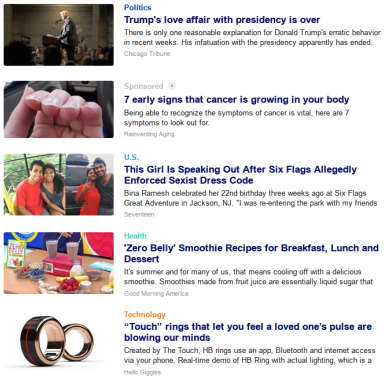

Figure 2.7: Sponsored content
Source: yahoo.com

Pop-up and pop-under ads open new browser windows, which can also cover the visible area of the screen or stay opened in the background until the main window is closed. The same principle also works with opening new tabs in the fore- or background. Some pages trick their users into clicking an ad by hiding it inside regular page elements such as the *Play* button of a video player. Ads can not only contain static images or text, but also flashing animations, video and audio. Furthermore ads cause additional network traffic and due to the complex processes of real-time auctioning can increase the loading time of a website significantly [69].

Research indicates that annoying ads are the main reason, why users turn to adblockers. In an experiment Goldstein et al. gave workers the task to categorize emails, which also contained ads with different levels of annoyance and payment levels (independent from

each other). This allowed them to create an economic model for estimating the cost of annoying ads [26] and showed that it was necessary to pay workers more for completing the same task with more annoying ads.

For researching how people actually use adblockers, a group of German researchers looked into a real data set of network traffic from the perspective of a European ISP and found that 22% of the most active users were browsing with *Adblock Plus* enabled [66]. Furthermore they were also able to determine parts of the client configurations, which indicates that most users were not as much concerned about protecting their privacy as they were about annoying ads.

Advertisers are facing new challenges with the current trend towards mobile computing. Processing power and data volume are limited resources on smartphones and tablets whereas those factors can be neglected on conventional PCs. Since many mobile apps use in-app advertising as their business model, adblockers are a threat to the app store ecosystem. In Apple's and Google's app stores, there are no apps available, which can block ads for the entire system because both platforms prohibit such apps in their terms of service. However there are at least browsers, which come with adblockers out-of-the-box (e.g. Eyeo's *Adblock Browser*) and some other browsers can be extended with plugins for adblocking. Solutions for blocking ads system-wide also exist but are only available in third-party app stores and some of them require root privileges on Android respectively jailbreak on iOS in order to function properly.

2.2.2 Privacy

One of the main concerns about online advertisement is its impact on the user's privacy. Advertising can directly and indirectly cause the leakage of personal information to third parties. The following points summarize the most common concerns.

Tracking

Advertisers' compensation usually increases with the *click-through-rate* (CTR) of the ads, which they feed to ad networks. In order to achieve high click-rates, ads are targeted to specific target groups or individual users. Trackers can identify users across websites and build profiles based on their browsing history and other personal information, which in turn aid estimating the effectiveness of an ad.

Similar to ad-tags, tracking scripts are included in the source code of a website and issue requests from the user's browser to tracking servers, when the page gets loaded. Some users are not comfortable with sharing their personal information with third parties. The *Do Not Track* (DNT) HTTP header is a mean for allowing users to signal websites not to track them [45]. As of now, the initiative has not succeeded and the DNT header is being ignored by the majority of advertisers. Furthermore state-of-the-art user tracking has evolved from cookie-based to fingerprint-based tracking, which not only recognizes a user with a mixture of technologies (Javascript, Flash, HTML5, etc.) but also the device he/she is using [52]. In comparison, fingerprint-based tracking is less transparent, as the

user is not able to delete cookies from the browser easily anymore. Even though browser fingerprinting raises the difficulty for dodging the radar of trackers, it can also be used for significantly improving session security by preventing session hijacking [81].

As trackers and advertisers have been lacking the willingness to make compromises in favor of user privacy, tracking blocking tools similar to adblockers were introduced in order to give users control over which tracking services to allow or block.

HTTPS

Over the last years the global usage of HTTPS has continuously increased. The leak of documents regarding NSA mass surveillance programs lead to a paradigm shift and users now have to assume that all of their network traffic is analyzed and stored by governments and ISPs all over the world. By encrypting data exchanged between browser and webserver, HTTPS can protect the user's privacy and increase the cost of bulk data collection. Although some certificate authorities such as *Let's Encrypt* now issue certificates free of charge and have streamlined the process of obtaining and renewing certificates, ad networks are still slowing down HTTPS deployment by providing only HTTP-based ad tags. The problem is, that browsers block requests to HTTP content on HTTPS pages, which consequently leads to ads not loading and therefor not being counted as impressions. To prevent revenue loss, many publishers wait until their current ad network supports HTTPS, before making the transition. This means that privacy and communication security are sacrificed on behalf of the user.

JavaScript code injection, in the form of a MITM attack, has been used by ISPs in the past to inject ads into HTTP traffic [5]. HTTPS can prevent code injections from MITM attacks, as the protocol verifies the integrity of data packets.

2.2.3 Malvertising

Malvertising is an umbrella term for malware delivered through ads. By either compromising infrastructure used in the ad delivery process or simply dodging quality checks, an attacker can insert malicious code into ads and have them embedded in legitimate websites. A user, who browses a website with embedded malicious ads, automatically runs this code. This code typically triggers a drive-by download, which can happen without the user noticing. In that case the code takes advantage of software vulnerabilities which often are present in browsers and their plugins. Exploits are also used to execute the downloaded content, which allows the attacker to deploy arbitrary malware on the victim's computer [41]. The same result can be achieved when the malicious embedded code impersonates a legitimate download such as a software update, which the user manually executes.

Malvertising per se is not a new phenomenon [65] but has become a popular attack vector. Multiple factors are responsible for that. On one hand the difficulty of malware development has decreased due to a growing number of easily available exploit kits. The trend towards mobile computing and proprietary black-box software also leaves

users exposed to vulnerabilities as security updates are still not delivered in a timely manner in many cases. Additionally the recent monetary success of ransomware gives attackers incentive to pursue those attacks. As malvertising offers the same possibilities for targeting victims as targeted advertising does, attackers can aim their malware at a selected target group, which in turn can further increase their profits. Events of the last years have shown that ad networks are not capable of delivering sufficient quality control and thus have lost many users' trust.

Incidents such as the malvertising campaign delivered through PageFair's CDN [4] and `forbes.com` [33] underline that forcing users to load ads can put them at risk and is there for unprofessional practice. This has been acknowledged by institutions such as the German BSI [22] and the Dutch NCSC [55], which recommend blocking ads for a secure browsing experience. Even Scott Cunningham from IAB in 2015 addressed the status quo of online advertising with the words *"we messed up"* in a statement [13] proposing to take the user's perspective more seriously.

2.3 Anti-Adblocking (AAB)

The term *anti-adblocking* is commonly used to describe all kinds of measures for preventing users from blocking ads. As there is no formal definition, it is often used synonymously with *adblock detection*. Typically an AAB solution is a script included in a website, which performs a check for determining whether or not ads are displayed and triggers an action such as displaying a pop-up, forwarding to another page or just silently tracking the user. The typical process for detecting and blocking AB users is described in the following steps:

1. **Add bait elements**

   For detecting active adblockers, AAB solutions add bait elements to the DOM of a website. Their purpose is to appear as regular ad and tracking elements, which trigger the adblocker's blocking mechanisms. This is done by requesting JavaScript files with suspicious names (e.g. ads.js, adsense.js) or inserting DOM elements with typical class names of ads (e.g. `pub_300x250`, `textAd`). Those names are blacklisted in most block lists, which results in the file or elements not being loaded when an adblocker is active. In the case of a JavaScript bait file, there usually is code included, which applies changes to a variable.

2. **Verify bait elements**

   In a second step the integrity of the bait elements is checked. In the case of a HTML elements, this is usually done by verifying that they are still present in the DOM, comparing CSS attributes with the initial values or searching for hiding properties as described in Listing 2.1. Bait scripts can be verified by setting variables and cookies or simply checking the contents of a function (e.g. character length).

3. **Response action**

   When the verification is not successful, meaning the bait has been tampered with, a response action is triggered. In many cases this is only a notification for the user, where the website asks him/her to whitelist or support it in another way (e.g. donations, subscriptions, micropayments, affiliate programs etc.). A more drastic measure is to prevent the user from consuming the website's contents similar to a paywall. For some websites it is also of interest to gather statistics about the adblocking-rate of their userbase, which can be done by reporting detection status back to a tracking server. Some websites also load different ads, which fulfill the *Acceptable Ads* guidelines [25].

AAB solutions can be categorized by their AB-detection mechanisms, their response action and their defenses against tampering and circumvention. The most common detection method is injection of bait elements as described in Step 1 and 2. Nevertheless side channels can also reveal the presence of adblockers. For example some adblockers expose resources [51] such as error pages, which can be checked. Methods against for protecting AAB scripts are explained in the Attack and Defense Model 3.1.

2.4 Anti-anti-adblocking

As a response to AAB solutions gaining prevalence, adblockers have also introduced countermeasures. There are several filter lists available, which serve the purpose of rendering AAB scripts useless and allowing users to display contents without being interrupted by them. Furthermore specialized solutions, which complement existing adblockers, have been introduced. To the knowledge of the author, the following solution is the most sophisticated one for this purpose.

Anti Adblock Killer (AAK) is a sophisticated open source solution for countering adblock detection scripts. It can circumvent more than 30 generic anti-adblocking solutions. It consists of a filter list, which must be used with an adblocking browser extension, and a UserScript. The syntax of the filter list is compatible with the most common adblocking extensions. The UserScript is JavaScript, which runs when a website is loaded and modifies its contents and scripts. It contains rules for generic anti-adblocking scripts as well as individual websites. Such rules typically perform tasks such as manipulating cookies, variables and parts of the page's original JavaScript code. Particularly useful is the `onBeforeScript` event listener because it allows to perform checks such as searching for variable names and strings within each individual script of the website. Listing 2.3 is an exemplary rule, which scans JavaScripts of the websites `tvspielfilm.de` and `finanzen.ch` for the string "UABPInject" and removes the script containing it. Filtering with such high granularity makes AAK a very effective complement for existing browser extension adblockers but the required setup and configuration are likely to prevent mainstream users from adopting this solution.

```
ad_defend_uabp : {
// note: when adblock detected inject new ads
// source: http://pastebin.com/cFQCp80W
host : ['tvspielfilm.de', 'finanzen.ch'],
  onBeforeScript : function () {
      return [{
        detected : 'AdDefend{UABPInject}',
        contains : 'UABPInject',
        external : false,
        remove : true
    }];
  }
},

ad_defend_uabp : {
// note: when adblock detected inject new ads
// source: http://pastebin.com/cFQCp80W
host : ['tvspielfilm.de', 'finanzen.ch'],
  onBeforeScript : function () {
    return [{
      detected : 'AdDefend{UABPInject}',
      contains : 'UABPInject',
      external : false,
      remove : true
    }];
  }
},
```

Listing 2.3: Example UserScript rule from *Reek's Anti-Adblock Killer*.
Source: `https://github.com/reek/anti-adblock-killer/blob/master/anti-adblock-killer.user.js`

CHAPTER 3

State of the Art

Research about online advertising, user tracking and its privacy impacts has been conducted and technical aspects of user tracking with cookies as well as browser fingerprinting are well explored. The economic view on the online advertising market and the value of privacy has also received attention from the scientific community. However, when narrowing the subject to literature related to detecting and blocking advertisements and trackers, until lately most effort went into understanding behavioral phenomenons of human interaction with ads and adblockers as well as tackling existing problems such as malvertising.

To the knowledge of the author the area of anti-adblocking software and adblock detection has only been covered by a small number of papers, which were published during the creation of this thesis.

3.1 Attacks against and Defenses of Adblocking

This section contains a model of the most common attacks against adblockers.

3.1.1 Obfuscation

Obfuscation aims at making code harder to understand for humans and prevent automated attacks. It is used to hide data and program logic from reverse engineering and prevent tampering. The following obfuscation methods are described by Xu et al. [84] in a paper focused on JavaScript malware obfuscation, but also apply to adblocking and AAB.

- **Randomization Obfuscation**

  Classical adblocking is signature based, meaning it relies on some kind of identifier (e.g. file name, domain or CSS selector) for detecting ads. It is possible to

randomize those identifiers on the server side, so that they change with every page load. Randomization can also be applied to AAB scripts and pop-up modals ("nag screens") to prevent automated detection. Randomization can also be performed on the client side for dynamically changing CSS selectors [1].

- **Data Obfuscation**

  Variables and constants can also be represented as the computational results of other variables and constants. *String splitting* splits a string into multiple substrings, which when concatenated result in the original string. The JavaScript functions `document.write()` and `eval()` can be used to execute concatenated strings. JavaScript also allows *keyword substitution*, which is arbitrary replacement of JavaScript keywords.

- **Encoding Obfuscation**

  JavaScript code can be encoded in different representations such as escaped ASCII characters, unicode, hexadecimal and custom encodings. Furthermore it is also possible to deliver encrypted code together with a decryption function and decrypt it during runtime.

- **Logic Structure Obfuscation**

  This technique changes the logic structure of the code by either inserting instructions not related to its actual functionality and/or adding conditional branches.

The average size of JavaScript files on websites has grown continuously through the last years [10]. On one hand this allows "hiding" small code snippets in plain sight because with the size of the JavaScript the search space for the snippet increases. On the other hand it has become common practice to use *minification* tools to reduce JavaScript file size by removing unnecessary characters and restructuring code. Depending on the methods used for minification, readability can suffer significantly. Minification can also combine multiple JavaScript files into one to minimize the number of HTTP(S) requests. By combining ads or adblock detection scripts with scripts actually required to deliver page content, it becomes more difficult to block them because it is no longer possible to block the request for the entire file (see also Section 5.3).

3.1.2 Defenses against Obfuscation

- **Blocking third-party content**

  Randomization can be countered in different ways depending on how it is applied. One of them is *domain rotation*, where ads are loaded from regularly changing domains. Classical adblocking tools would have to add all of those domains to their filter lists and deliver them to the user immediately when they start being used. This would require high maintenance effort and bloat their filter lists. In response to the company *Yavli* actually practicing domain rotation, Easylist has made a

statement[18]. It says that all third-party content will be blocked on websites using Yavli. This effectively not only blocks ads, but also all third-party analytics and tracking scripts.

- **JavaScript deobfuscation in the JavaScript engine**

  Even if obfuscation has been applied in a way that makes the source code of a script unreadable to humans, at some point it will be compiled and executed by the JavaScript engine. The *JavaScript Deobfuscator* [61] is a developer tool, which displays code as it is being executed, at which point it is already deobfuscated. This can be very useful for understanding AAB scripts and detecting a script if it is randomly obfuscated on page load but the core functionality is always the same.

- **Static code analysis and simulated execution**

  Many of the above mentioned obfuscation techniques can be defeated precalculating the results of functions with static results. Tools such as *JSDetox* [78] furthermore emulate script execution and a HTML DOM environment.

3.1.3 Escalation of the Arms Race

As various obfuscation techniques still pose challenges for comparing JavaScript code snippets, AAB detection based on code analysis faces the same problems as JavaScript malware detection. *Abstract syntax trees* (AST) [14] are a form of code representation, which allows to overcome some parts of obfuscation such as variable naming. Mughees et al. also analyzed the identified AAB scripts based on their AST representations and were able to cluster them by the similarity of their structure. They showed that most observed first-party AAB scripts are in one dense cluster and therefor share high structural similarity. Nevertheless, as with malware, it is possible that new, structurally different scripts are not detected by comparing them against the already known clusters of AAB scripts.

The paper by Storey et al.[1] [77] introduces an analytical framework and new techniques for demonstrating how ads and AAB scripts can be defeated in the future. They developed a state model which categorizes scenarios where either publisher or user "wins" (see Figure 3.1).

State 1: User has no effective adblocker - ads are displayed.
State 2: User successfully blocks ads by using an adblocker, which is effective at detecting ads.
State 3: Publisher detects adblocker and is able to deny access to content.
State 4: User successfully blocks ads and AAB software.

[1] not yet published in a scientific journal

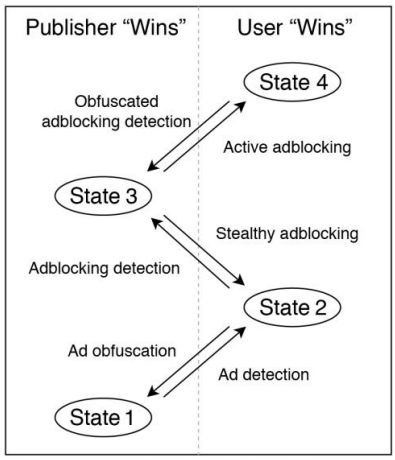

Figure 3.1: State model of adblocking conflict
Image source: Storey et al. [77]

The model considers measures taken by the user or publisher as transitions between those states. Based on it, Storey et al. proposed new techniques for demonstrating the means available to users, which are summarized in the following paragraphs.

- **Perceptional adblocking** is a technique for detecting ads by taking visible features into account. It is based on the assumption that ads have to be labeled as ads to adhere to legal requirements. Instead of static filter rules, as used by common adblocking extensions, perceptional adblocking detects ads as objects in the DOM by considering multiple criteria during runtime. In a first step all containers with dimensions typical for ads are selected. Fuzzy image matching and searching is then used to detect icons contained in disclosure standards such as the common AdChoices [2]. Furthermore optical character recognition (OCR) is performed[2] to identify text such as "Sponsored" in images. A click simulator follows links in hidden browser tabs to determine if they are related to an ad disclosure standard. By those criteria it is determined, whether a container contains an ad or not. The main advantage of this approach is its ability to overcome ad obfuscation. The paper contains a case study stating that perceptional adblocking is very effective against Facebook ads, which are not easily distinguishable by their CSS `id` and `class` attributes.

- **Rootkit-style stealthy adblocking** is supposed to defeat adblock detection. It creates a new DOM root node above the original one and injects a subtree with overlay containers, which cover the detected ads. All of the website JavaScript pointing to the root node needs to be replaced, so that it points to the original root

[2]The paper states that OCR performs badly and has only minimal impact on the effectiveness of perceptional adblocking.

node. The DOM traversal API is modified so the website JavaScript can't reach the new root node or the subtree of overlay containers. This is possible due to the higher privilege level of extension JavaScript, which allows hiding extension data from website JavaScript. The website CSS is modified to only apply to the original DOM subtree. This method has the drawback, that the website JavaScript can detect interception of property access.

- **Shadow execution style stealthy adblocking** creates a shadow copy of the unmodified DOM and maps DOM elements in the original tree to their counterparts in the shadow copy. The shadow copy tree remains unmodified whereas adblocking is applied to the original tree. API calls are executed on both DOM subtrees but the responses are only returned from the shadow copy. This way from the perspective of the website JavaScript it is impossible to detect CSS changes (such as hiding ads). On the downside *shadow execution style stealthy adblocking* has not yet been implemented, as it is would require changes on the browser source code. Furthermore it might be possible for publishers to detect duplicate requests originating from the same host.

- **Signature-based active adblocking** aims at blocking ads despite active AAB scripts on a website. Active adblocking in this context is a mixture of HTTP request and response manipulation and modification of JavaScript. Not all browsers allow HTTP response manipulation, and thus also require a proxy. Similar to classical filter list adblocking, lists of signatures are created by humans, who manually inspect the website's source code. The signature lists can consist of regular expressions (e.g. function and variable names), structural JavaScript signatures or semantic signatures using call graphs. The paper is accompanied by an implementation of a signature based active adblocker with regular expression-based signatures effectively circumventing AAB scripts on 49 websites.

- **Differential active adblocking** is a theoretical technique for detecting AAB code. The website gets loaded twice - once without any manipulation and once with an active adblocker. By "visual diffing" the impact of AAB scripts (e.g. popup) are identified. By comparing the execution traces, it becomes possible to identify the code paths responsible for the visual changes. Finally the responses of API calls made from those code paths in the adblocking-enabled version of the website are replaced by the responses from the unmodified version.

The presented techniques are very promising for visually blocking ads, however they should not be seen as a panacea. With (partial) exception of signature-based active adblocking, they only address the problem of annoying ads, but still leave users vulnerable to malvertising and tracking, as untrusted JavaScript still is executed on the user's client. Furthermore they don't reduce network traffic because all ads are downloaded before detecting them. In some cases network traffic might even increase because a website gets loaded twice. Furthermore there is a performance impact (+0.53 seconds of page loading

time for perceptional adblocking without OCR on an Intel i7 powered laptop). Therefor it is questionable if it will also be usable on slower mobile devices with limited battery power.

3.1.4 Attack and Defense Trees

For this thesis the following models were created with the modeling tool *ADTool* [36] to visualize the ongoing struggle between adblockers and anti-adblockers. They each represent the transitions between two stages from the state model by Storey et al. (see Figure 3.1).

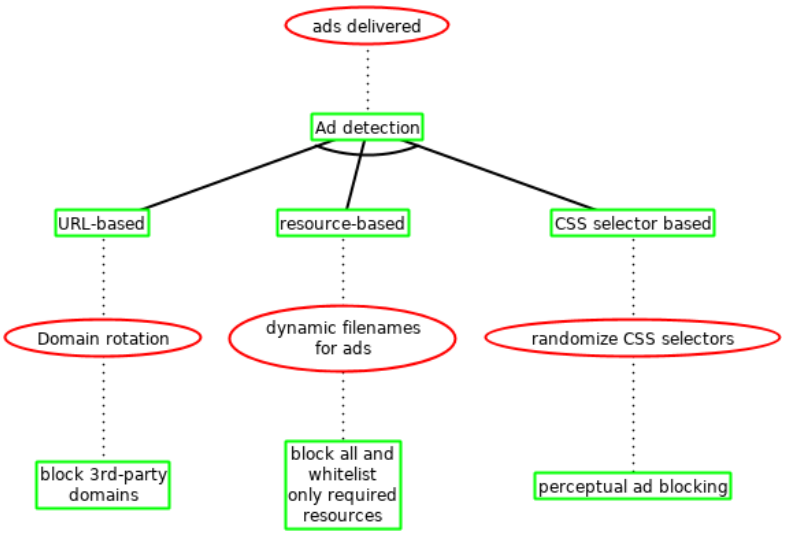

Figure 3.2: Attacks and Defenses between stage 1 and 2

Figure 3.2 shows the transition between state 1 and 2. Adblockers can detect ads by three criteria. A domain or a pattern contained in the `URL` (e.g. `/ads`) can be used block resources. Publishers can try to circumvent such static filter rules by using random domains to deliver their ads. This can temporarily unblock ads, but in the past filterlist providers have reacted to this tactic by simply blacklisting all requests to third party domains and creating exception rules for resources that are required for the main functionality of the website. If ads are delivered from the same origin as the content, adblockers can still block the resources by their file name or patterns in the relative path (e.g. `ads.js`). In this case the adblocker can block all resources and again only whitelist desired resources - this time based on resource patterns instead of domains. The third case requires that an ad has already been loaded and is injected into the DOM of a website. Adblockers can detect ads based on their CSS selectors such as `class`, `id`, and their parent and sibling nodes. By changing their CSS attributes they can make them invisible to the user (see Listing 2.1). If the publisher randomizes CSS selectors, ads can no longer be distinguished from real content. In this case perceptional adblocking can be used to identify ads.

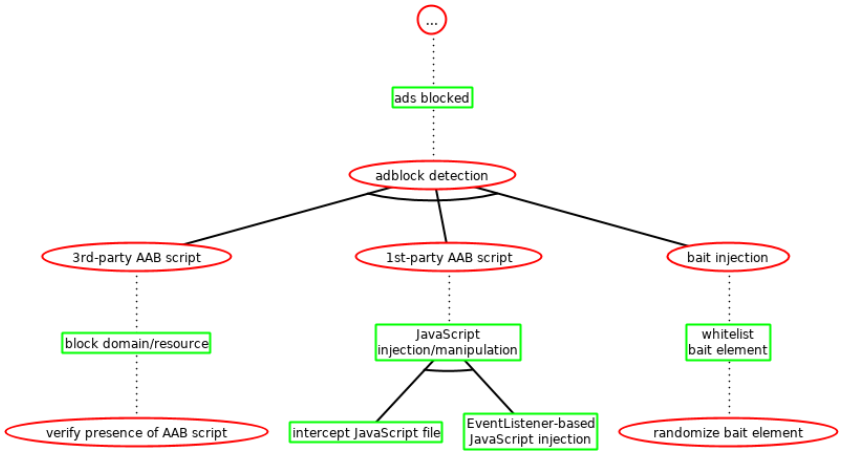

Figure 3.3: Attacks and Defenses between stage 2 and 3

Figure 3.3 starts with the scenario that ads have been successfully blocked. Publishers can now try to run AAB scripts in the user's browser. AAB scripts loaded from third party services can be blocked with URL filter rules the same way ads are blocked. To prevent this, checks to verify that the AAB script was loaded can be placed in the source code of the page. If an AAB check takes the form of inline-JavaScript, it can still be blocked by intercepting API-calls. It is possible to block functions by their name or search for patterns such as variable names or strings contained in them and block execution or inject a replacement function if necessary. If the bait element has static identifiers between page loads, it can be whitelisted with an exception rule. Publishers can randomize identifiers of bait elements to prevent that.

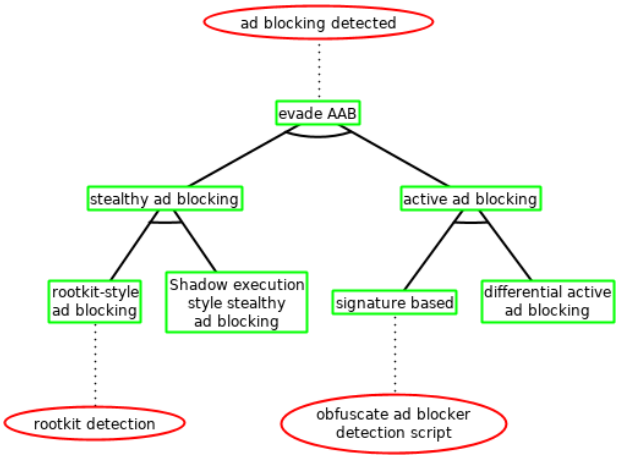

Figure 3.4: Attacks and Defenses between stage 3 and 4

Lastly Figure 3.4 shows how adblockers can evade adblock detection without preventing execution of AAB scripts. For details see Section 3.1.3.

3.2 AAB detection

Two papers use webcrawlers for detecting anti-adblocking scripts but they use different approaches.

Mughees et al. use automated A/B testing in combination with machine learning to detect whether a site uses anti-adblocking scripts [51]. They do so by first loading the page multiple times without any active adblockers - this allows them to identify what the content of the website should look like when it is fully accessible and noise created by dynamic content can be filtered out. The crawler loads the same page with an adblocker enabled and then compares the two pages for changes in the DOM nodes (added/removed nodes, visibility property changes, etc.). In the test set (1200 websites) the approach detects the existence of anti-adblockers with a precision of 94.8%. On the downside this method does not automatically detect which anti-adblocker was used - information that is of essence for countering the anti-adblocker.

Nithyanand et al. [53] go one step further by not only testing if an anti-adblocking script is present, but also identifying the scripts. They first crawl a list of websites and download each embedded or linked JavaScript in them. By comparing each JavaScript to all other JavaScripts with vector multiplication (Term Frequency–Inverse Document Frequency) they measure the similarity and can identify scripts which are basically the same with some minor alterations (e.g. website-ID) and group them into *cliques*. One downside of this approach is that the runtime $O(n^2)$ increases heavily with the number of JavaScripts n, which imposes limitations for scaling to a large number of websites. The second problem is that it is still necessary to manually investigate whether a *clique* is actually a family of anti-adblocking scripts or is unrelated to adblocking. Furthermore it is possible to obfuscate JavaScript, in such way that its true functionality only becomes visible during execution.

3.3 Tracking

User tracking is the foundation of targeted advertising and therefor an integral component of online advertising. Furthermore many tracking blockers are based on the same methods as adblockers. Popular tools such as *Ublock Origin* and *Adblock Plus* even combine those features and provide filter lists for both purposes.

Merzdovnik et al. performed a large scale study among the Alexa 200,000 top ranked pages, where they analyzed the effectiveness of tracker-blocking tools for browsers and mobile devices.
The paper differentiates between three types of rulesets. *Community-driven* rulesets are blacklists maintained by a number of contributors in public repositories, whereas *centralized* rulesets are maintained by companies. Opposed to them *algorithmic* rulesets such as the one used by EFF's Privacy Badger's are generated on the client by a heuristic. *Ghostery*, which uses a centralized ruleset, blocked the most third party requests compared to other popular ad and tracker blockers.

Furthermore the study looked into performance impact of those blocking tools. The researchers found that the tested browser extensions did not significantly increase CPU load - in the cases of *uBlock Origin* and *Disconnect* it even got reduced. The initial memory consumption was higher with than without an extension, but after crawling 30 webpages all extensions except *ABP* reached a similar or even lower level of memory usage than the plain browser. [47]

A framework for categorizing and detecting 3rd party tracking already exists even though due to the fast evolution of tracking, the information might not be up to date [72].

The effect of adblocker usage with regards to different industries has been investigated in 2006 [39] and is also reported annually by the company *Pagefair*[58, 59, 60] - a company which offers sophisticated anti-adblocking software. They claim, that some websites such as gaming- and technology-related ones suffer from significantly higher adblock-rates than others, which underlines that there is no single best strategy for handling the threat of decreasing revenues due to adblocking.

Goldstein et al. created an economic model for estimating the cost of annoying ads [26] and the paper *Follow the Money* by Gill et al. quantifies the value of information about users and comes to the conclusion that even small numbers of users blocking trackers lead to large revenue drops for publishers [24].

Pujol et al. looked into a real data set of network traffic from the perspective of an European ISP and found that 22% of the most active users were browsing with *Adblock Plus* enabled. Furthermore they were also able to determine parts of the client configurations, which indicate that most users were not as much concerned about protecting their privacy as they were about annoying ads.[66]

3.4 Latest developments

The continuously increasing demand for adblocking did not go unnoticed. As stated in the 2017 Pagefair report mentioned earlier, the user numbers of adblockers have increased continuously through the last years (see Figure 1.1). According to it the combined number of desktop a mobile adblocker users has increased from 326 million in January 2015 to 616 million in December 2016 (+ 89,0%). The increase of mobile AB user numbers is amplified by the general trend towards using mobile devices for browsing, where absolute growth numbers are much higher than on desktops.

The *Coalition for Better Ads* is a industry-driven project, which defined standards[21] for less intrusive ads similar to *Eyeo*'s *Acceptable Ads* program. Major players of such as Google, Facebook and the IAB joined the coalition. Google announced to include an adblocker into Chrome [68], which blocks ads not complying with those standards and is enabled by default. According to StatCounter, Chrome has a market share of more than 50% [76] on both desktop and mobile devices. Therefore this step will most likely impact the ad industry heavily. Smaller ad networks and publishers will be forced to also join the coalition and adhere to their standards in order to avoid significant losses.

Apple's iOS 9 introduced a content blocking API for Safari on mobile devices. Third party apps can use this to block ads, trackers and inappropriate contents. The company also announced to include a tracker blocker in the desktop version of Safari to protect their privacy [83]. It will use an algorithmic approach, where the ruleset is generated on the client using machine learning for classifying, which domains have the ability to track the user cross-site. Only cookies and website data from domains the user actively visits (first-party context) are stored. For one day they can also be used in 3rd-party context and after 30 days without revisiting a domain they are purged.

Facebook has engaged in a cat and mouse game against the adblocking community. It removed CSS selectors, which were prior used by adblockers for ad detection and replaced it with regularly changing CSS classes. As a result the ads became almost impossible to distinguish from real content in Facebooks "timeline". This method is only possible because Facebook acts as publisher and ad network at the same time, which means that resource blocking is not effective due to the ads being served from the same domain.

Critics of AAB software argue that the advertising industry has not fixed the problem of malvertising and thus disabling adblockers poses a security threat. Assuming that malvertising campaigns aim at infecting as many machines as possible, publishers with AAB scripts are better attack channels than those who tolerate adblockers. Incidents of malvertising campaigns[12] on websites, which are using anti-adblocking software (e.g. `forbes.com` [33] and pagefair [4]) imply that the topics malvertising and anti-adblocking software are closely related and such attacks might become more common in the future.

CHAPTER 4

Methodology

This chapter describes the research methods chosen to answer the research questions. The first part is aimed at understanding how AAB tools work and what the strengths and weaknesses of different techniques for AB-detection are. The second part aims at automatic detection of AAB scripts on websites.

4.1 Literature review

The first step was to search for existing literature and review it for relevance to the topic. This includes scientific literature as well as online resources, software documentation and source code of existing adblocking related tools. In this step a list of the most common adblockers was created (see Section 2.1.2).

4.2 Case study

The case study takes a look at the AAB implementation of three representative websites and their effects. The websites `bild.de`, `wired.com` and `forbes.com` were chosen because they have large user bases and are pioneers in applying AAB software (in comparison to websites with similarly high user numbers). We analyzed the websites' implementations of anti-adblocking scripts and compared their adblocking detection methods and their defenses against tempering with them. For experimenting with those websites the latest available version of *Mozilla Firefox* [49] on *Fedora* [70] was used. In a first step each website was visited with only *Adblock Plus* and default settings enabled to learn how the website reacts. Next JavaScript was disabled completely with the extension *NoScript* to determine if the page's contents can be accessed. Furthermore the scripts responsible for adblock detection were identified by blocking individual scripts with *NoScript* and *uMatrix*. Then we looked into the source code of the identified scripts and used developer tools provided by *Firefox* [50] together with *JavaScript Deobfuscator* [61]

to monitor the DOM and understand obfuscated code. Furthermore existing methods for countering the AAB scripts were analyzed and we tested which of the following adblock and content filter solutions were able to bypass the anti-adblocking scripts:

- **NoScript** was chosen because it disables JavaScript and other scripts completely. It also offers an option to ignore `<noscript>` elements, which can be used to detect the presence of the extension, as is the case on `bild.de`.
- **Ghostery** is an anti-tracking tool with a large number of users. It was configured to only block all trackers from the *Advertising* and *Adult Advertising* category.
- **Adblock Plus (ABP)** represents the de facto standard for adblocking as it is the most widely used adblocking extension. It provides an option for allowing unintrusive ads (see Section 2.1.2) and for subscribing to the *Adblock Warning Removal List (AWRL)* by *Easylist*.
- **uBlock Origin** is a popular alternative to *ABP*, which consumes less system resources and does not compromise on unintrusive ads. It can use filter lists with the same syntax as ABP and by default besides its own filter list has *Easylist* and other third party lists enabled.
- **Anti Adblock Killer (AAK)** was the most comprehensive solution for countering AAB software at the time of writing, to the knowledge of the author. In table 5.1 *Reeks AAK* means that both the filter list and the userscript provided on the AAK website were enabled.

The effectiveness of those pioneer AAB solutions is important because other content providers, who are considering to use anti-adblockers themselves, will base their revenue expectations on the success or failure of them. Therefor we also researched user statistics and how they developed after the introduction of AAB. Because real user numbers are hard to obtain, the Alexa rank was used as an indicator. It should be noted, that Alexa ranks are not evenly distributed and therefor the numbers can not be compared directly.

4.3 Webcrawler

This section aims at generating a model for automatically detecting the presence of AAB scripts on websites. For this purpose a webcrawler was implemented to collect data from a list of websites. By loading a website multiple times with different browser profiles, it is possible to detect differences in its representation. Mughees et al. published a methodology for detecting AAB scripts based on monitoring of DOM changes such as nodes added/removed, text changes and style changes [51]. They performed A/B testing and used those features to train machine learning algorithms and build a model. A random forest classifier was most successful and reached a precision of 94.8%. My implementation follows a similar approach but focuses on optical changes visible to humans, when they see a website on a screen.

4.3.1 Candidate Generation

Candidates are tuples consisting of a `URL` and a `crawlMode`. A `crawlMode` is a browser profile which contains a configuration for using a specific browser extension. For each `URL` multiple of those tuples are generated, each representing a `crawlTask`. For training the machine learning algorithm a list of 50 domains was retrieved from the *Adblock Warning Removal List* [17] and *Anti-Adblock Killer* [71] filterlists. Additionally 50 domains were randomly selected from the Alexa top 1 million sites. For all of them candidates were generated once with a plain browser profile and once with *ABP* configured to block *Acceptable Ads* as this is one of the most common scenarios for triggering an AAB script. The crawler then took screenshots of all those pages once per profile, which were manually evaluated. This way they were categorized as `positive` (AAB detected) and `negative` (no AAB detected).

crawlMode	Extension
`van`	**Vanilla**: No browser extensions enabled
`abp`	**Adblock Plus**: Acceptable Ads disabled
`ubo`	**uBlock Origin**: Default settings
`gho`	**Ghostery**: *Advertising* and *Adult Advertising* lists blocked

Table 4.1: Descriptions of `crawlModes`

To avoid interference between websites (e.g. cookies) for each page visit a new browser instance with a fresh profile is created. This also helps achieving higher stability and keeps the memory footprint of the crawler small. To make sure that an AAB is actually triggered, the crawler simulates user interaction by scrolling up and down randomly for 20 seconds after loading the page. It then returns to the top of the page and takes a screenshot. During this time it monitors the current browser `URL` and logs redirects if they occur. This is necessary because some AAB scripts react delayed and would not be visually recognizable without this waiting time. Also pop-ups and browser notifications can unpredictably disturb the screenshots. Therefor browser notification dialogues were disabled with the following setting:

```
1 # disable notification dialogues
2 self.profile.set_preference('dom.webnotifications.enabled', False)
```

Even though there also are cases where only on subpages of a website AAB scripts are used, we limited the candidate `URL`s to the landing pages of websites[1]. Including subpages is out of scope for this thesis, as it would drastically increase runtime. Instead of intrusive

[1] with the exception of the AAB demo pages

AAB warnings, some publishers only make subtle changes to their pages, which are not visible without further interaction with the website. For example a publisher can block parts of the website's functionality such as video playback or a comment section when no AB was detected. Furthermore AAB scripts can also silently set cookies or only report the AB detection to a tracking server. Detecting those subtle changes is also out of scope for this thesis.

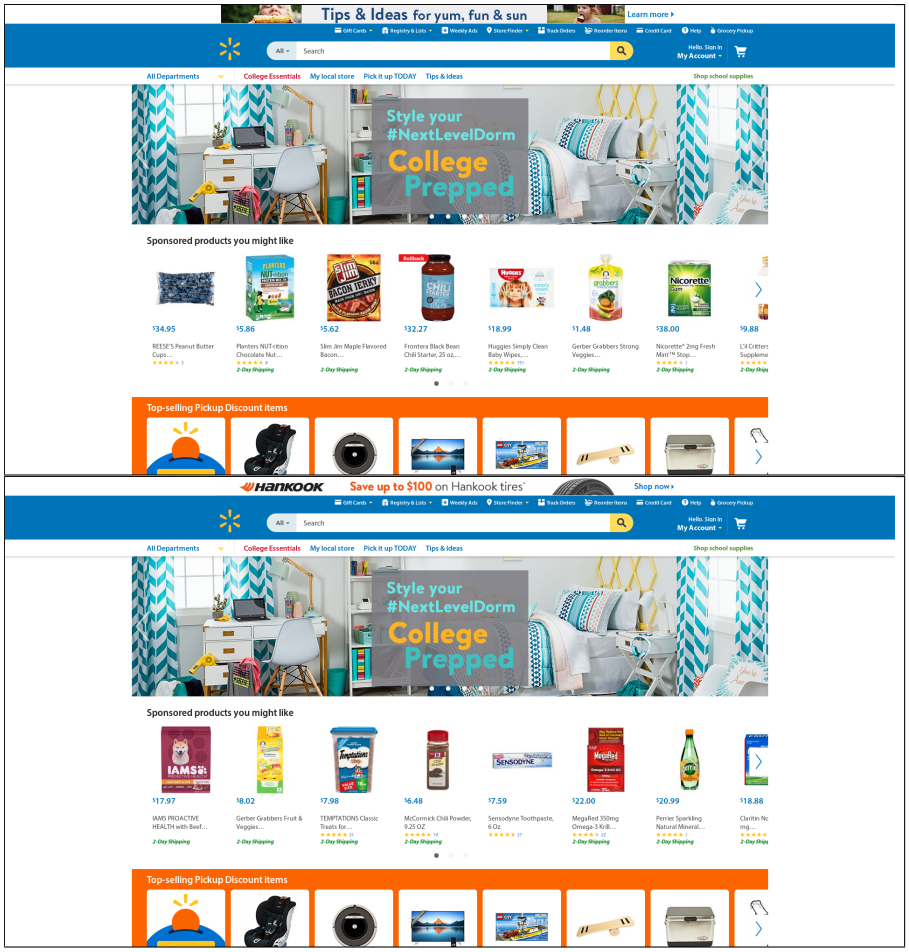

Figure 4.1: Loading `wallmart.com` twice yields different screenshots due to dynamic content

Many websites have dynamic contents (e.g. ads, videos, newsfeeds), which create visual differences between visits. Since only the differences between candidates with different browser profiles are of interest, dynamic contents can be considered as noise. To be more robust against that, the crawler visited each candidate `URL` multiple times with each browser profile.

4.3.2 Anti-Adblock Detection

AAB scripts can be detected on multiple levels. Firstly they must either be present in the source code of a website or loaded by another script during runtime. Secondly AAB scripts apply changes to the DOM when they inject bait elements and perform counteractions such as changing content visibility or adding a pop-up container. Some AAB solutions use redirects, which only occur when an adblocker is detected.

Mughees et al. [51] already showed, that AAB detection based on the observation of changes to the DOM is possible and can yield good prediction results. Unfortunately they did not release their crawler implementation. Recreating their experiments for improving them further would have been too much effort for the scope of this thesis.

The approach chosen for this thesis therefor relies on a completely different set of features. It is based on the assumption that AAB scripts create visual changes to a website, when they detect an AB.

Feature extraction

For each `crawlTask` a screenshot is taken and redirects are logged. Screenshots of the same website can be compared against each other by measuring visual differences. Besides the logging of redirects, all features are extracted from screenshots taken by the crawler.

Visual similarity matching

For measuring visual similarity, the python library *imagehash* [9] is used. It calculates hashes, which can be seen as fingerprints of images. The Hamming distance [31] between two hashes is a measure for visual differences between the images. A Hamming distance of 0 means that the images are identical, greater values mean greater differences. The library offers multiple perceptional hashing algorithms, of which `pHash` [38] and `dHash` [37] are used in this thesis. The library was chosen because its algorithms are designed to perform well on large datasets as its original purpose is to detect duplicates of pornographic images and prevent them from being accepted when uploading to a platform. The two hashing algorithms work as follows:

Masking

To anticipate the noise produced by dynamic content, we introduce the concept of *masking*. The idea is to identify the parts of hashes, which vary even between screenshots produced with the same browser profile. A mask has the same number of fields as a hash (64) with values between 0 and 1 in them. This number is the average of the bit values, considering them as 0 or 1. A value close to 0.5 means that the bit values at the same position of the hashes have high variance. The idea is that mask values can indicate noise, that should be ignored. To generate a mask at least three image hashes are necessary. Table 4.3 exemplifies the generation of a mask from three hashes of images taken with `crawlMode van`. Masking is an enhancement on top of `pHash` and `dHash` as the hashes don't need any modification to be compared with masking. For comparison a threshold

pHash	dHash
1. **Reduce resolution** to 32x32 pixels 2. **Reduce to greyscale** 3. **Compute *discrete cosine transformation*** (DCT) - 32x32 values 4. **Reduce DCT** to top left 8x8 values (low-frequency values) 5. **Compute average DCT** value of all DCT values except first term 6. **Reduce DCT values** to 64 bit: If the DCT value is above average set bit to 1, else to 0 7. **Construct hash**: Generate 64 bit integer number from the reduced DCT values as hash	1. **Reduce resolution to 9x8** pixels 2. **Reduce to greyscale** 3. **Calculate differences** between adjacent pixels 4. **Reduce to bits**: if the left pixel is brighter than the right adjacent pixel, set to 1 else to 0 5. **Construct hash**: Generate 64 bit integer number from bit values

Table 4.2: Sequence of steps performed by pHash and dHash algorithms

is required, which defines how close to 0.5 values can be before they are regarded as noise. When comparing two hashes with masking, the following check is performed for each position of the hash:

```
1 if abs(val - 0.5) < self.maskTolerance:
2   #set both bits at position n to 0
3   imgHash1.hash[x][y] = 0
4   imgHash2.hash[x][y] = 0
```

This code replaces the bits at position n of local copies of both compared hashes with 0 when the value of position n in the mask is within $0.5 \pm threshold$. This way, when the Hamming distance is calculated the bits at position n are ignored.

Figure 4.2 shows how groups of screenshots are compared to each other. The following steps are taken to calculate similarity per website:

1. **Generate similarity values of vanilla candidates.**
   Similarity values for all screenshots taken with `crawlMode van` are calculated. In Figure 4.2 they are the represented by the blue edges $\{sA_1, sA_2, sA_3\}$. For later comparison both `pHash` and dHash are used with and without masking.

vanHash1

1	1	1	1	1	1	1	1
1	1	1	1	0	0	0	0
0	0	0	0	0	0	0	0
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
0	0	0	0	1	1	1	1
1	1	0	0	1	1	0	0
0	0	0	1	1	0	0	0

vanHash2

1	1	1	1	1	1	1	1
1	1	1	1	0	0	0	0
0	1	0	0	0	0	0	0
1	0	1	0	1	0	1	0
0	1	0	1	1	1	0	1
0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0
0	0	0	1	1	1	0	1

vanHash3

1	1	1	1	1	0	1	1
1	1	1	1	0	0	0	0
0	1	0	0	0	0	0	0
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
0	0	0	0	0	0	1	1
1	0	0	0	1	1	0	0
0	0	0	1	1	1	0	1

Mask

1	1	1	1	1	0.67	1	1
1	1	1	1	0	0	0	0
0	0.67	0	0	0	0	0	0
1	0	1	0	1	0	1	0
0	1	0	1	0.33	1	0	1
0	0	0	0	0.67	0.67	1	1
1	0.33	0	0	1	1	0	0
0	0	0	1	1	0.67	0	0.67

Table 4.3: Example mask generation from three hashes. Red cells are ignored for a threshold > 0.16

2. **Calculate mean of `van` similarity values.**
   The mean of all values (per hashing algorithm) generated in Step 1 is calculated as an indicator of normal noise on the page.

3. **Generate similarity values between vanilla and adblock candidates.**
   For each screenshot with `crawlMode van` is compared against each screenshot with `crawlMode abp`. In Figure 4.2 the red edges $\{sB_1, sB_2, sB_3, sB_4, sB_5, sB_6, sB_7, sB_8, sB_9\}$ represent the similarity calculations between the two candidate sets.

4. **Calculate mean of `abp` similarity values.**
   The mean of all values (per hashing algorithm) generated in Step 3 is calculated and denotes the average difference between an `van` screenshot and a `abp` screenshot.

Optical Character Recognition (OCR)

A second visual feature, which differs between visiting a website with `crawlMode van` and `abp` is the visible text. Mughees et al. found that the DOM features, which contributed the most information gain to their AAB classifier were *change in number of words* (35.44%), *change in number of text nodes* (27.89%), *change in number of lines* (18.17%), *numbers of changes in nodes* (17.37%) *and change in numbers of characters* (17.19%) [51]. Text-related features appear to have significant impact on AAB detection. Since most this information is not only contained in the DOM but also in website screenshots, we decided to try extracting it. The python library `pytesseract` [32], which is based on the *Tesseract* OCR engine [75], was used. To obtain better results, the images were preprocessed to greyscale and thresholding was applied. However OCR is not originally designed to operate on images with noisy, colorful backgrounds and varying font sizes. Even though overall text is not very well recognized, metrics such as the number of characters can still be valid indicators for similarity. Furthermore adblock warnings are typically placed at well readable positions of the website and often contain similar text. Therefor pattern matching was also used to count the occurrences of patterns such as the word "adblock". OCR offers the advantages that adblock warnings can also be detected if they are displayed as an image file and is furthermore resistant against obfucation of the text in the website's source code. For each website the mean values of character count and pattern occurences count are calculated for each browser profile.

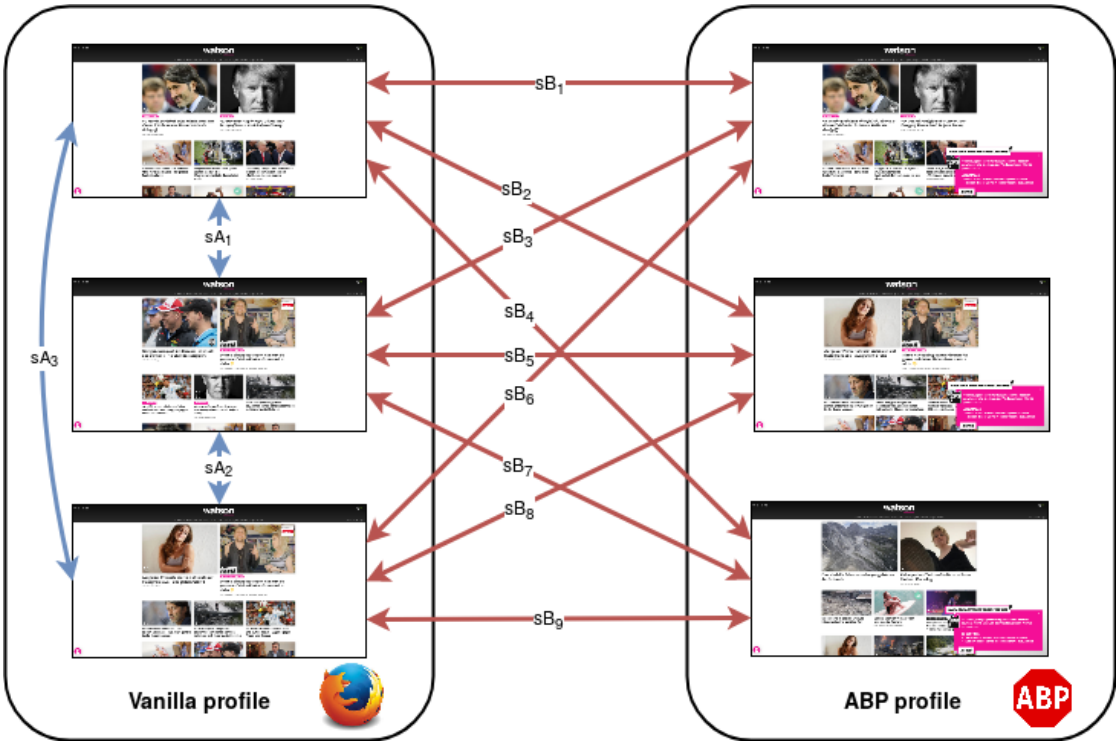

Figure 4.2: Calculation of visual similarity values for screenshots of `watson.ch`. All `van` screenshots are compared to each other (blue edges) and each `abp` screenshot is compared to each `van` screenshot (red edges).

Feature Aggregation

In a post-processing step the features are combined to express the differences between two browser profiles. All `PercentageDiff` values were calculated with an offset of 1 to avoid empty values due to division by 0. Table 4.4 explains how they are calculated.

Features	
redirectDiffVan	Average number of redirects which are different between page visits with `crawlMode van`
redirectDiffVanAbp	Average number of redirects which are different between page visits with `crawlMode van` and `abp`
redirectDiffOfDiffVanAbp	Absolute difference between `redirectDiffVanAbp` and `redirectDiffVan`
similarityMeanVan	Average visual similarity value between page visits with `crawlMode van`
similarityMeanVanAbp	Average visual similarity value between page visits with `crawlMode van` and `abp`
similarityDiffVanAbp	Absolute difference between `similarityMeanVanAbp` and `similarityMeanVan`
similarityPercentageDiffVanAbp	Percentual increase or decrease between `similarityMeanVanAbp` and `similarityMeanVan`
ocrCharCountVan	Average number of characters with `crawlMode van`
ocrCharCountAbp	Average number of characters with `crawlMode abp`
ocrCharCountDiffVanAbp	Absolute difference between `ocrCharCountAbp` and `ocrCharCountVan`
ocrCharCountPercentDiffVanAbp	Percentual increase or decrease between `ocrCharCountAbp` and `ocrCharCountVan`
ocrPatternMatchesVan	Average number of pattern occurrences with `crawlMode van`
ocrPatternMatchesAbp	Average number of pattern occurrences with `crawlMode abp`
ocrPatternMatchesDiff	Absolute difference between `ocrPatternMatchesAbp` and `ocrPatternMatchesVan`
ocrPatternMatchesPercentageDiff	Percentual increase or decrease between `ocrPatternMatchesAbp` and `ocrPatternMatchesVan`

Table 4.4: Feature descriptions

AAB Classification

For generating predictive models about AAB presence on a website,the *H2O* machine learning tool suite [30] was used. Experiments were conducted with various combinations of features (see Table 4.4) and the following classification algorithms were applied:

- Naive Bayes
- Distributed Random Forest (DRF)
- Deep Learning

Evaluation

All models were evaluated with k-fold cross-validation, where $k = 10$. To examine the effectiveness of the different visual similarity algorithms, the original dataset was split into the following 5 datasets.

Datasets	
dHash	All redirect- and OCR-related features, `dHash` similarity values
pHash	All redirect- and OCR-related features, `pHash` similarity values
dHashMask	All redirect- and OCR-related features, `dHash` similarity values with masking
pHashMask	All redirect- and OCR-related features, `pHash` similarity values with masking
combined	All redirect- and OCR-related features, `dHash` and `pHash` similarity values with and without masking

Table 4.5: Features contained in datasets

CHAPTER 5

Case Study

In this chapter the websites `bild.de`, `forbes.com` and `wired.com` are analyzed. All three of them are news platforms with large user bases and pioneers of anti-adblocking. The case study looks into how their anti-adblocking scripts work, how effective they are for actually locking out adblock users and how their introduction impacted their businesses.

5.1 Bild.de

Bild.de is an online news and entertainment platform belonging to the publisher *Axel Springer SE*. As one of the first of the major German websites, it has been running an anti-adblocking script to enforce that their ads are displayed. The anti-adblocking software manifests itself by redirecting to the URL `http://www.bild.de/wa/11/bild-de/unangemeldet-42925516.bild.html` when an active adblocker is detected.

Figure 5.1: Bild.de detected an Adblocker

This URL is hard-coded into the main HTML-file as embedded JavaScript (see Listing 5.1). Apparently the 8 digit numbers are codes for differentiating between subscribers, adblocker users and regular users. They are not randomized and the URL strings are not obfuscated or encrypted. This makes it easy to detect the script and prevent its execution for adblockers capable of interfering with inline JavaScript - All they have to do is search for the referrer URL string in all external JavaScript files and inline JavaScript and stop them from running. However, all blockers working only with per-file or per-host granularity have difficulties because the relevant

JavaScript is included in the main HTML file of each page, which is necessary to load. Host-based adblockers also can be defeated with the adblock detection script for the same reason. Even though strings are not obfuscated, the program logic is.

The first hurdle to overcome is a job offer, which is displayed in the JavaScript console (see Figure 5.2).

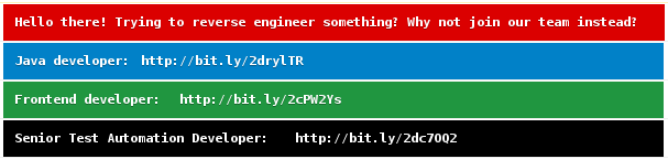

Figure 5.2: Job offers in JavaScript console on `bild.de`

```
a : {
  a : 1,
  b : {
    a : '42925672',
    b : 'http://www.bild.de/wa/ll/bild-de/bildplus-42925672.bild.html',
    c : '42925678',
    d : 'http://www.bild.de/wa/ll/bild-de/angemeldet-42925678.bild.html',
    e : '42925516',
    f : 'http://www.bild.de/wa/ll/bild-de/unangemeldet-42925516.bild.html'
  },
  c : '43645300;43645254;42925650;42925604;42925636;42925628;42925622;
      42925602;42925618;42925614;42925608',
  d : 2
},
```

Listing 5.1: Hard-coded referrer URL in embedded JavaScript. Source: `www.bild.de`

A bait element is loaded from `cdn1.smartadserver.com` - a domain, which is present on all major adblocking blacklists (see Listing 5.2). It introduces a boolean variable with the value `true`.

```
var sasverify=true;
```

Listing 5.2: Bait element *verify.js*

The inline script in Listing 5.3 loads an image file from `http://atsfi.de` and appends a parameter to the GET request. If the bait script has been executed the parameter takes the value `0` and if `sasverify` is `undefined` and the value is `1`.

```
<script type="text/JavaScript">
(function(){
  document.createElement("img").src="http://atsfi.de/s.png?b="+
```

```
4     ((typeof(sasverify) == "undefined")?1:0);
5   })();
6   </script>
```

Listing 5.3: Inline JavaScript on `bild.de` sends tracking status disguised as image

An obfuscated jQuery script from `code.bildstatic.de` verifies the integrity of the anti-adblocking scripts. The function `getAbData()` calls `g()`, which in turn calls `f()` to verify that *SmartAdServer* has been loaded correctly. Because the file has a size of 331 KB and consists of mostly unreadable code, it was necessary to use the Firefox extension Javascript Deobfuscator, which allows to inspect code generated on the fly as it is compiled and executed by the JavaScript engine [61].

```
1   getAbData: function() {
2     return g()
3   }
4
5   ...
6
7   function g() {
8     return n === !1 && (n = {
9       elemHidden: !1,
10      saDisabled: f()
11    }, n.ba = n.elemHidden || n.saDisabled), n
12  }
13
14  ...
15
16  function f() {
17    return (void 0 === window.SmartAdServerAjax ||
          window.SmartAdServerAjax.toString().length < 50) && void 0 ===
          window.sasmobile
18  }
```

Listing 5.4: Deobfuscated jQuery verification script from `code.bildstatic.de`

Listing 5.5 and Listing 5.6 are the filter rules and userscript from AAK[71]. The filter rules are written in the syntax of ABP. They define that no cosmetic filtering is performed on `bild.de` and scripts from the domains `sascdn.com` and `smartadserver.com` are whitelisted.

```
1   ! bild.de
2   ! https://github.com/reek/anti-adblock-killer/pull/687
3   @@||bild.de^$elemhide
4   @@||sascdn.com^$script,domain=bild.de
5   @@||smartadserver.com^$script,domain=bild.de
```

Listing 5.5: Filter rules from *Reek's Anti-Adblock Killer* filter list

The userscript searches every function on `bild.de` for the redirect URL string and replaces it with `JavaScript:undefined`.

```
1  bild_de : {
2  // issue: https://github.com/reek/anti-adblock-killer/issues?q=bild
3  host : ['bild.de'],
4    onBeforeScript : function () {
5        return [{
6        contains :
             'http://www.bild.de/wa/ll/bild-de/unangemeldet-42925516.bild.html',
7        external : false,
8        replace : ['JavaScript', 'void(0);'].join(':')
9      }];
10   }
11 }
```

Listing 5.6: UserScript script from *Reek's Anti-Adblock Killer*

Axel Springer SE engaged in an court battle with Eyeo, claiming that the *Acceptable Ads* initiative was illegal. They also threatened to sue video blogger Tobias Richter[23] under the claim that his video tutorial about circumventing the anti-adblocker was infringing a German national law, which prohibits circumvention of copyright protections ([1])[15]. This argumentation lacks accuracy because neither the website's contents are encrypted nor does the anti-adblocking script serve the purpose of protecting them from being copied - it serves the purpose of protecting ad impressions.

5.2 Forbes.com

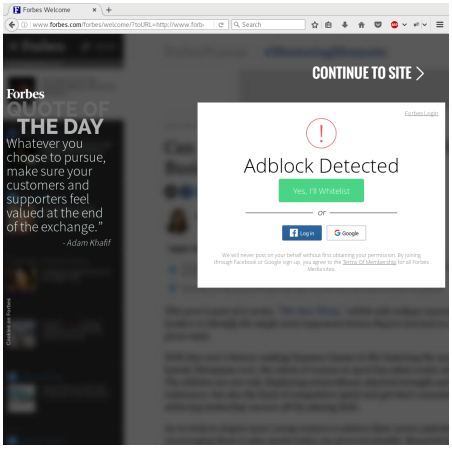

Figure 5.3: `Forbes.com` gateway page

The business magazine *Forbes* introduced an AAB script around December 2015. Similar to `bild.de`, the user is redirected to a page to notify him/her to turn off adblocking. When no adblock is detected, the cookies `dailyWelcomeCookie` and `welcomeAd` are set to `true`. Appearently the check is only performed, when requesting the index page or when the two cookies are not set. The script `1714dc93.main.js` from `i.forbesimg.com/welcomead/scripts/` (see Listing 5.7) seems to be responsible for detecting adblockers. Particularly interesting is the function `setBypassCookie()` because - as the name suggests - it sets the cookies for bypassing the adblock detection. Since those cookies are all that is required to bypass the detection script it is possible to set them manually. The only caveat is that they have an expiration date (12:00 AM next day), which means they have to be set again after a few hours.

```
1  function() {
```

[1] German translation: "technische Schutzmaßnahme für urheberrechtlich geschützte Inhalte"

```
2   "use strict";
3   WelcomeAd.Modules.AdBlockChecker = Backbone.View.extend({
4     checks: [],
5     first: !Cookies.get("forbes_ab"),
6     checking: !1,
7     adblock: !1,
8     initialize: function(a) {
9       _.assign(this, a)
10    },
11
12    addCheck: function(a, b, c) {
13      var d = this;
14      this.checks.indexOf(a) > -1 || (this.checks.push(a), b && "number" ==
            typeof b && setTimeout(function() {
15        d.checks.indexOf(a) < 0 || (d.triggerAdBlockState(!(c === !1)),
              d.removeCheck(a))
16      }, b), 1 === this.checks.length && this.check())
17    },
18
19    removeCheck: function(a) {
20      this.checks.indexOf(a) > -1 &&
            this.checks.splice(this.checks.indexOf(a), 1)
21    },
22
23    check: function() {
24      this.checking = !0;
25      var a = this,
26      b = setInterval(function() {
27        if (a.adblock || 0 === a.checks.length) return a.checking = !1, void
              clearInterval(b);
28        for (var c = 0; c < a.checks.length; c++) a.checks[c] &&
              a.checks[c]() && a.removeCheck(a.checks[c])
29      }, 10)
30    },
31
32    triggerAdBlockState: function(a) {
33      this.adblock || (this.adblock = a, a &&
            this.trigger("AdManager:AdBlockDetected"))
34    },
35
36    getAdBlockState: function() {
37      return this.adblock
38    },
39    ...
40
41    setBypassCookie: function(a) {
42      var b, c, d = new Date(+new Date + 2592e6),
43      e = Cookies.getJSON("global_ad_params") || {};
44    e.ab = {
45      value: "off",
46      expiration: d.getTime()
47    }, b = _.isUndefined(urlParams.force_ab) ? Cookies.get("forbes_ab") :
          urlParams.force_ab.toUpperCase(), c = b && b[0].match("A") ? b[0] :
```

```
        "A",  Cookies.set("forbes_ab", c, _.merge({
48        expires: d
49      }, this.cookie.settings)), Cookies.set("global_ad_params", e, _.merge({
50        expires: d
51      }, this.cookie.settings)), a ? (Cookies.set("welcomeAd", !0, _.merge({
52        expires: d
53      }, this.cookie.settings)), Cookies.set("dailyWelcomeCookie", !0,
          _.merge({
54        expires: d
55      }, this.cookie.settings))) : (Cookies.remove("welcomeAd",
          this.cookie.settings), Cookies.remove("dailyWelcomeCookie",
          this.cookie.settings))
56    },
```

Listing 5.7: `js_options.js` loaded from `http://i.forbesimg.com/welcomead/scripts/1714dc93.main.js`

Alternatively to setting the cookies, it was also possible to deactivate the Adblocker only for the URL `http://www.forbes.com/forbes/welcome/`, because only here the checks are performed. This implies that it was possible to browse the website without being exposed to ads.

AAK adapted a userscript rule just a few days after forbes had introduced the anti-adblocking feature (see Figure 5.8). It sets the two cookies mentioned before to **`true`**, updates their expiration dates and navigates the browser to an actual destination URL stored as the cookie `toUrl`. This ensures that the user gets redirected away from the detection page.

```
1  forbes_com : {
2    // by: Giwayume
3    // issue: https://github.com/reek/anti-adblock-killer/issues/865
4    host : ['forbes.com'],
5    onStart : function () {
6      if (window.location.pathname.indexOf('/welcome') > -1) {
7        Aak.setCookie('welcomeAd', 'true', 86400000, '/');
8        Aak.setCookie('dailyWelcomeCookie', 'true', 86400000, '/');
9        window.location = Aak.getCookie('toUrl') || 'http://www.forbes.com/';
10     }
11   }
12 },
```

Listing 5.8: `forbes.com` rule from AAK list [71]

5.3 Wired.com

Figure 5.4: Bild.de detected an Adblocker containing `blockadblock` is loaded.

The technology news website `wired.com` uses the open-source script *blockadblock* [74]. Compared to the other two websites it is less intrusive because it does not display its message for disabling adblockers immediately. Instead the detection script gets triggered after about 1 minute and only when page scrolling is detected, which at least gives the reader enough time to decide if the content is worth whitelisting `wired.com`. Since *BlockAdBlock* is a generic script also used by other websites, there are already filter rules for circumventing it available.

Listing 5.9 shows that first a boolean variable `blockadblock` is set to **`false`** and then a script `load`

```
<script > var blockAdBlock = false; < /script>
...
<script src="http://www.wired.com/assets/load?scripts=true&c=1&load%5B%5D=
outbrain,blockadblock,tracking,wired,ads,wp-embed&ver=1470953511"
    type="text/JavaScript"> < /script>
```

Listing 5.9: Script tag for loading *blockadblock* on `wired.com`

Listing 5.10 shows the most relevant functions from the BlockAdBlock script. First class names for creating bait elements and their CSS attributes are defined. The class names are the same as in the example configuration on the BlockAdBlock website. The bait elements are injected as `div` tags by the `b.prototype.createBait` function. `b.prototype.checkBait` then verifies that they are still present and their CSS attributes have not changed. The check is performed in a loop and bait elements are removed after each iteration.

```
! function(a) {
  var b = function(b) {
    this._options = {
      checkOnLoad: !0,
      resetOnEnd: !1,
      loopCheckTime: 50,
      loopMaxNumber: 5,
      baitClass: "pub_300x250 pub_300x250m pub_728x90 text-ad textAd text_ad
          text_ads text-ads text-ad-links",
      baitStyle: "width: 1px !important; height: 1px !important; position:
          absolute !important; left: -10000px !important; top: -1000px
          !important;",
      debug: !1
    },
    ...
```

```
13      b.prototype._creatBait = function () {
14        var b = document.createElement('div');
15        b.setAttribute('class', this._options.baitClass),
16        b.setAttribute('style', this._options.baitStyle),
17        this._var.bait = a.document.body.appendChild(b),
18        this._var.bait.offsetParent,
19        this._var.bait.offsetHeight,
20        this._var.bait.offsetLeft,
21        this._var.bait.offsetTop,
22        this._var.bait.offsetWidth,
23        this._var.bait.clientHeight,
24        this._var.bait.clientWidth,
25        this._options.debug === !0 && this._log('_creatBait', 'Bait has been
              created')
26      },
27      ...
28      b.prototype.check = function (a) {
29        if (void 0 === a && (a = !0), this._options.debug === !0 &&
              this._log('check', 'An audit was requested ' + (a === !0 ? 'with
              a' : 'without') + ' loop'), this._var.checking === !0) return
              this._options.debug === !0 && this._log('check', 'A check was
              canceled because there is already an ongoing'),
30        !1;
31        this._var.checking = !0,
32        null === this._var.bait && this._creatBait();
33        var b = this;
34        return this._var.loopNumber = 0,
35        a === !0 && (this._var.loop = setInterval(function () {
36          b._checkBait(a)
37        }, this._options.loopCheckTime)),
38        setTimeout(function () {
39          b._checkBait(a)
40        }, 1),
41        this._options.debug === !0 && this._log('check', 'A check is in
              progress ...'),
42        !0
43      },
44      b.prototype._checkBait = function (b) {
45        var c = !1;
46        if (null === this._var.bait && this._creatBait(), (null !==
              a.document.body.getAttribute('abp') || null ===
              this._var.bait.offsetParent || 0 == this._var.bait.offsetHeight ||
              0 == this._var.bait.offsetLeft || 0 == this._var.bait.offsetTop ||
              0 == this._var.bait.offsetWidth || 0 ==
              this._var.bait.clientHeight || 0 == this._var.bait.clientWidth) &&
              (c = !0), void 0 !== a.getComputedStyle) {
47        var d = a.getComputedStyle(this._var.bait, null);
48        ('none' == d.getPropertyValue('display') || 'hidden' ==
              d.getPropertyValue('visibility')) && (c = !0)
49        }
50        this._options.debug === !0 && this._log('_checkBait', 'A check (' +
              (this._var.loopNumber + 1) + '/' + this._options.loopMaxNumber + '
              ~' + (1 + this._var.loopNumber * this._options.loopCheckTime) +
```

```
          'ms) was conducted and detection is ' + (c === !0 ? 'positive' :
          'negative')),
51      b === !0 && (this._var.loopNumber++, this._var.loopNumber >=
          this._options.loopMaxNumber && this._stopLoop()),
52      c === !0 ? (this._stopLoop(), this._destroyBait(), this.emitEvent(!0),
          b === !0 && (this._var.checking = !1))  : (null === this._var.loop
          || b === !1) && (this._destroyBait(), this.emitEvent(!1), b === !0
          && (this._var.checking = !1))
53    },
```

Listing 5.10: `blockadblock` script on `wired.com`

The rule for countering from *AAK* is also a generic solution which is designed to work for all websites using *BlockAdBlock*. It injects multiple JavaScript functions to replace the original AAB script.

```
1  fakeFuckAdBlock : function (instanceName, className) {
2
3      // inject fake fuckadbluck
4      Aak.addScript(Aak.intoString(function () {
5
6        var CLASSNAME = function () {
7          var self = this;
8          var callNotDetected = false;
9          this.debug = {
10           set : function () {
11             return self;
12           },
13           get : function () {
14             return false;
15           }
16         };
17         this.onDetected = function (callback) {
18           this.on(true, callback);
19           return this;
20         };
21         this.onNotDetected = function (callback) {
22           this.on(false, callback);
23           return this;
24         };
25         this.on = function (detected, callback) {
26           if (!detected) {
27             callNotDetected = callback;
28             setTimeout(callback, 1);
29           }
30           console.info(['AntiAdbKiller', location.host, 'FuckAdBlock']);
31           return this;
32         };
33         this.setOption = function () {
34           return this;
35         };
36         this.options = {
37           set : function () {
38             return this;
```

```
39            },
40            get : function () {
41              return this;
42            }
43          };
44          this.check = function () {
45            if (callNotDetected)
46            callNotDetected();
47          };
48          this.emitEvent = function () {
49            return this;
50          };
51          this.clearEvent = function () {};
52        };
53
54        Object.defineProperties(window, {
55          CLASSNAME : {
56            value : CLASSNAME,
57            writable : false
58          }
59        });
60
61        Object.defineProperties(window, {
62          INSTANCENAME : {
63            value : new CLASSNAME(),
64            writable : false
65          }
66        });
67
68      }).replace(/INSTANCENAME/g, instanceName || 'fuckAdBlock')
69      .replace(/CLASSNAME/g, className || 'FuckAdBlock'));
70
71  },
```

Listing 5.11: Userscript for replacing `BlockAdBlock` on `wired.com` from *AAK* list [71]

5.4 Comparison

Effectiveness of AAB solutions
For evaluating the effectiveness of the three AAB solutions, different browser configurations were used (see Section 4.2) to browse the pages and determine if they would be detected as adblock users (see Table 5.1).

NoScript, as the most primitive method, was only able to bypass `wired.com` with default settings. With the `hide <noscript> elements` option enabled, it also made content on `bild.de` available without showing ads. However blocking all scripts also means that dynamic content such as image galleries, videos or third-party feeds are not displayed correctly. Even though JavaScript is a de facto standard of web development, publishers should keep in mind that completely locking out NoScript users can negatively impact their ranking in search engines[7].

	bild.de	wired.com	forbes.com
NoScript@default*	blocked	OK	blocked
NoScript + hide <noscript>*	OK	OK	blocked
Ghostery@Advertising**	blocked	blocked	OK
ABP@default*	blocked	blocked	blocked
ABP + acceptable ads*	blocked	blocked	blocked
ABP + AWRL*	blocked	blocked	blocked
uBlock Origin@default*	OK	blocked	OK
ABP + Reek's AAK*	OK	OK	OK
uBlock O. + Reeks AAK*	OK	OK	OK

** tested on November 16, 2016
* tested on February 20, 2017

Table 5.1: Adblock detection matrix

Adblock Plus was detected by all three websites with default configuration as well as with *AWRL* and *acceptable ads* enabled. AWRL only contains cosmetic filters for removing HTML elements such as `div` tags. A possible explanation is that ABP, due to its large user base, is a main target of AAB software.

With default configuration *uBlock Origin* was only detected and blocked by `wired.com`, which makes it the most effective out-of-the-box adblocking solution without noteworthy caveats.

Anti-Adblock Killer successfully unblocked all three websites with both *ABP* and *uBlock Origin*.

Impact on user base
Publishers turn to AAB solutions for increasing their revenue. Whether or not this expectation is realistic depends on many factors such as the adblock rate (% of users with active adblockers), target audience and uniqueness of the contents. Measuring the adblock rate not trivial due to the many different methods for adblocking and the ongoing efforts to evade adblock detection. Furthermore this information is rarely published and can only be speculated on. However *Alexa* provides website statistics including a global traffic ranking, which can be seen as an indicator for user numbers. Figure 5.5 shows how the user numbers reacted when the AAB solutions were introduced to the three websites. The 2017 Pagefair report[60] found in a survey that 74% of users confronted with an *adblock wall* would rather leave the website than disable their adblockers. All three case study websites confirm this tendency as all of them have increased their Alexa ranking, which reflects a decrease in the number of visitors.

`bild.de` introduced its AAB solution after an upwards trend in user numbers which for the next 4 months fell significantly. The Alexa rank had its peak in September 2015 (257.5) but then fell drastically until it stabilized in March 2016 (414.61). After August 2016 a second downwards trend in user numbers followed which lead to a low point in

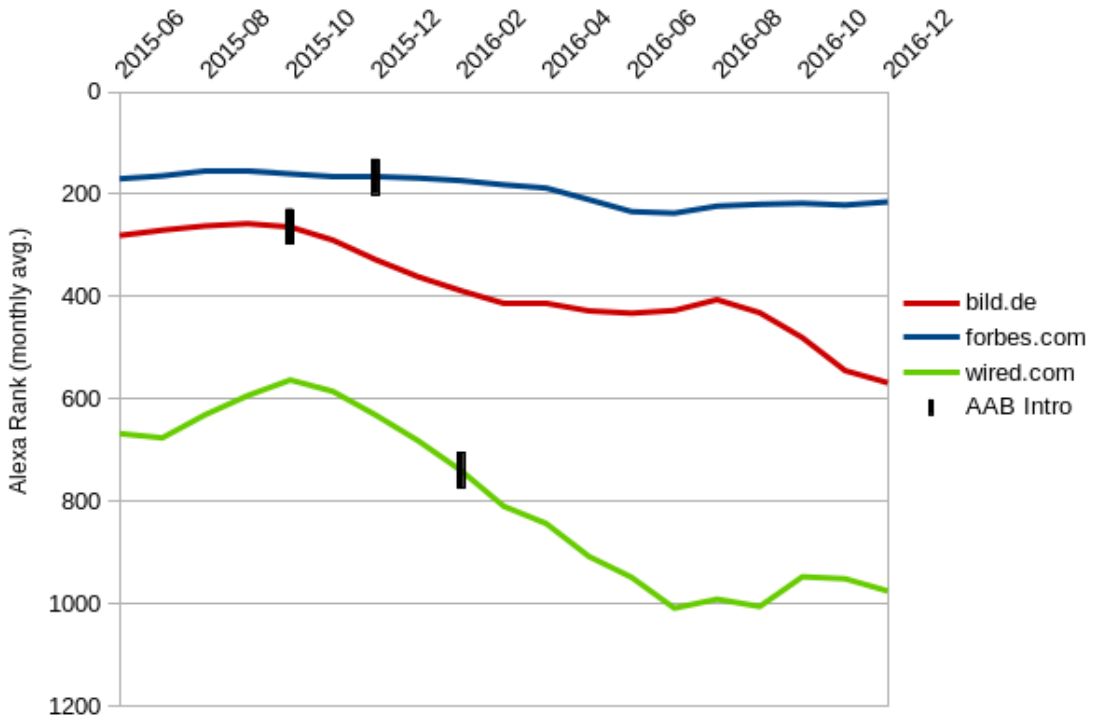

Figure 5.5: Alexa rankings over time (monthly averages)

Website	AAB introduction
`bild.de`	10/2015
`forbes.com`	12/2015
`wired.com`	02/2016

Table 5.2: AAB introduction dates

December 2016 (568.52).

`forbes.com` had a ranking of 155.23 in September 2015. After the AAB introduction in December 2015 (166.58) the website lost visitors until July 2016 (237.0). The ranking then stopped falling and improved slightly and remained at 214.94 in December 2016.

`wired.com` started losing visitors in October 2015 (563.0) and introduced its AAB solution in February 2016 (739.86). The user numbers kept decreasing until July 2016 (1008.87), where they stagnated shortly and even increased in October 2016 (947.45).

Overall the websites share a clear trend of declining user numbers immediately after AAB introduction. According to their own statements[16][28], `forbes.com` and `bild.de` have 42 respectively 67 percent success rates in making their users turn off their adblocker. However even though adblock users do not directly contribute to the publishers revenue streams, they still can benefit them in many ways such as sharing articles with other users (some them will visit the page without adblocker) or creating user content. Furthermore as

a result of outrage in the adblocking community, efforts for boycotting websites with AAB solutions were made by the influential social news aggregation website `reddit.com`[6] and others.

Regarding AAB solutions, the Pagefair adblock report 2017 came to the verdict:

"Adblock walls are ineffective at motivating most adblock users to disable their adblock software, even temporarily. Unless the website in question has valued content that cannot be obtained elsewhere, an adblock wall is likely to be ineffective at combatting adblock usage at any significant rate."[60]

Month	**bild.de**	**forbes.com**	**wired.com**
2015-06	280.70	169.87	667.80
2015-07	270.32	164.26	676.19
2015-08	262.00	155.87	631.10
2015-09	257.50	155.23	593.57
2015-10	264.00	160.16	563.00
2015-11	289.73	165.27	585.30
2015-12	327.90	166.58	631.39
2016-01	361.42	168.55	681.87
2016-02	388.59	173.45	739.86
2016-03	414.61	181.55	809.65
2016-04	414.77	187.90	843.57
2016-05	428.00	210.58	907.77
2016-06	432.50	234.00	948.60
2016-07	427.23	237.00	1,008.87
2016-08	406.06	223.32	991.29
2016-09	431.57	219.77	1,005.63
2016-10	480.45	217.77	947.45
2016-11	544.67	221.17	951.30
2016-12	568.52	214.94	975.39

Table 5.3: Monthly average of Alexa rankings

CHAPTER 6

Software Design

This chapter describes the design of the implemented software.

6.1 Software Architecture

The implemented software consists of the following components:

- Candidate generator
- Crawler
- Visual Similarity Comparer
- OCR Reader
- Result Verifier
- Result Aggregator

The entire project is written in Python [67] and was tested with version 3.6[1] on Fedora 26 [70]. For better scalability the performance critical components, namely *Candidate Generator*, *Visual Similarity Comparer* and *OCR Reader* are loosely coupled. To achieve this, the distributed task queue *Celery* [11] with a *RabbitMQ* 3.6.10 server [63] as its backend was used in version 4.1.0. This enables parallelization of tasks, because Celery can spawn multiple worker threads simultaneously. Furthermore the program could easily be extended to distribute `crawlTasks` between multiple computers by changing the RabbitMQ host configuration to point to the same queue.

[1] Most components also work with Python 2.7, but the OCR reader requires a Python 3 environment.

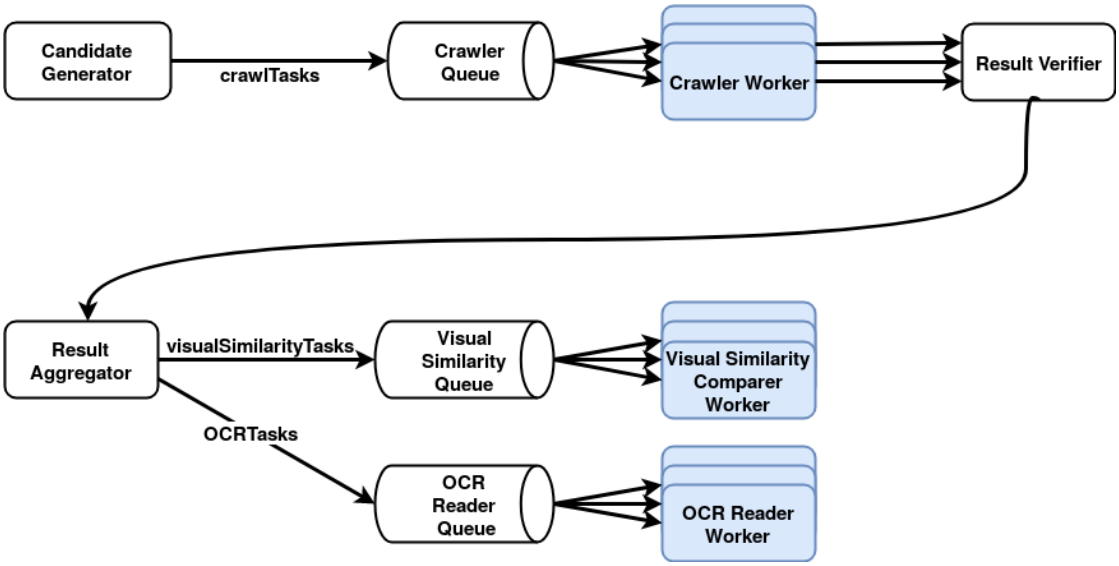

Figure 6.1: Software Architecture

6.2 Candidate Generator

The candidate generator takes a list of URLs as input, which it then parses. For each of the URLs a number of `crawlTasks` is generated in accordance to a global configuration file. In the configuration file the number of crawlTasks per `crawlMode` is defined. Masking (see Section 4.3.2) requires at least three screenshots with `crawlMode van`. In order not to flood the websites with multiple requests at once, the list of URLs is traversed multiple times, once per crawlMode, so that sequential `crawlTasks` do not contain the same URL.

```
1 # crawlerConfig.py
2 ...
3 VAN_RUNS = 3  # Vanilla profile
4 ABP_RUNS = 1  # Adblock Plus Profile
5 ...
6 # urlList
7 example1.com
8 example2.com
```

Listing 6.1: Example of `crawlerConfig.py` and an `urlList`

With the configuration from Listing 6.1 the tasks would be enqueued in the following order:

	URL	crawlMode
1	example1.com	van
2	example2.com	van
3	example1.com	van
4	example2.com	van
5	example1.com	van
6	example2.com	van
7	example1.com	abp
8	example2.com	abp

Table 6.1: Example sequence of `crawlTasks`

6.3 Crawler

The Crawler aims at triggering as many AAB scripts as possible by mimicing a real human user and therefor needs to run with an authentic configuration. *Selenium Webdriver* [73] is a software-testing framework for browser automation. It is compatible with multiple popular browsers and was used to control *Mozilla Firefox* [49]. The open-source privacy analysis framework OpenWPM [19] uses a similar technology stack, but only supports Firefox until version 48, which collides with the goal of mimicing a real user and might also lead to incompatibilities with browser extensions. The Crawler can be run in headless mode, which is based on the *Xvfb virtual framebuffer X server for X* [82]. A `CrawlerWorker` instance is started as a Celery task and is and is only used for one task. At the begin of the task a new browser profile is created with Webdriver and depending on the `crawlMode` parmeter an extension is loaded. While executing a `crawlTask`, mouse scrolling is simulated and the current browser `URL` gets checked for changes. At the end of the crawl, a screenshot is taken and the list of `URL` changes and a reference to the screenshot file is saved as a JSON [8] results file. Both files are stored in a directory named after the domain of the website - multiple `crawlTasks` therefor write to the same directory.

6.4 Result Aggregator

With the results from the `crawlTask` four operations need to be performed:

- Verify results from `crawlTasks`
- Generate visual similarity values
- Perform OCR scans
- Aggregate results

The Result Aggregator creates Celery tasks to perform those operations and enqueues them to worker queues. This way they are run in parallel. The tasks are performed on per website, which means that for example a `visualSimilarityCompareTask` gets the domain `example1.com` as parameter and then loads all images in the directory `example1.com` to compare them. When all result files from the workers are present, the Result Aggregator generates a summary of all features (see Table 4.4). Comma-separated values (CSV) file, which can be used as the dataset for the machine learning experiments.

6.5 Result Verifier

For the analysis of captured screenshots, it is a prerequisite that the results from the `crawlTask` are verified. The Result Verifier serves the purpose of control that during a `crawlTask` both a result file and a screenshot have been persisted. This is necessary because the Webdriver instance is terminated at the end of a `crawlTask`. During high system load, I/O operations can be too slow to persist the files before Celery terminates the thread. The Result Verifier checks that every referenced screenshot actually is present in the file system and the other way around that every screenshot in the file system is referenced in a results file. All files not fulfilling those criteria are moved to a hidden directory and excluded from further analysis. While iterating over the results files, the Results Verifier also compares the lists of redirects contained in them. For that results are compared pairwise and the number of disjunct items is counted as an indicator of the difference in redirects. This has the advantage that dynamic redirects, meaning that the browser is always redirected to a dynamically changing `URL`, are also detected when comparing the Crawler's results with `crawlMode van` and can be interpreted as noise.

6.6 Visual Similarity Comparer

For each screenshot in a website's directory hashes are generated and compared as described in Section 4.3.2. Hashes are persisted to avoid having to recalculate them, when analyzing visual similarity multiple times. While performing the comparison operation, the program first checks whether hash files for the given screenshots already exist and generates them otherwise. The results are again stored in a JSON file.

6.7 OCR Reader

The OCR Reader scans screenshots and tries to extract the text contained in it (see also Section 4.3.2). We observed that sometimes incorrect whitespaces were present in the parsed text, which in turn could impede pattern detection. Therefor all whitespaces were removed from the text and furthermore the capitalization of all characters was changed to lower case. OCR results are stored in a JSON file.

CHAPTER 7

Evaluation

7.1 Candidate Generation

Originally, the training dataset consisted of 50 pages with AAB (14 with strict and 36 soft blocking pages) and 50 without. In total 11 pages were not loaded correctly (timeout or HTTP status code 404 error) often enough to have enough samples to apply masked visual similarity comparison. The Alexa top 1 million `URL` list was obtained in February 2017 and is not published anymore. The actual crawls were performed in August and September 2017, therefor it makes sense that some of the websites do not exist anymore. The unreachable websites were excluded from the dataset, which now consisted of 89 (44 AAB positives and 45 AAB negatives). All screenshots were manually inspected to verify that they had been correctly classified.

7.2 Visual Similarity Algorithms

To evaluate the effectiveness of `dHash`, `pHash` with and without masking, models were trained with the H2O machine learning suite [30]. The five datasets described in Table 4.5 were trained with *Naive Bayes*, *Distributed Random Forest* and *Deep Learning* classifiers and k-fold cross-validation, where $k = 10$.

	dHash	pHash	dHashMasked	pHashMasked	combined
AUC	90.12%	90.07%	**92.36%**	89.88%	90.29%
Precision	84.96%	83.17%	**86.82%**	81.17%	84.22%
Recall	85.39%	85.60%	**85.61%**	84.85%	84.85%

Table 7.1: Evaluation of Visual Similarity algorithms. Averages of all classifiers

Table 7.1 contains the Area under ROC Curve (AUC), precision and recall as standard measures of prediction quality. Overall the results were very close to each other.

dHashMasked performed slightly better than the others and achieved an average precision of 86.82%. The effectiveness of all dHashMasked models was among the best results with all classifiers.

Classifier	Dataset	Precision	Recall	AUC
Deep Learning	dHashMask	94.74%	81.82%	93.94%
Distributed Random Forest	dHash	84.27%	85.71%	92.12%
Distributed Random Forest	dHashMask	81.63%	90.91%	91.67%
Naive Bayes	All	82.22%	84.09%	91.57%
Naive Bayes	dHashMask	84.09%	84.09%	91.46%
Naive Bayes	pHash	82.61%	86.36%	91.31%
Naive Bayes	pHashMask	82.98%	88.64%	91.31%
Naive Bayes	dHash	84.09%	84.09%	91.26%
Distributed Random Forest	pHashMask	79.59%	88.64%	90.86%
Distributed Random Forest	All	84.09%	84.09%	90.43%
Distributed Random Forest	pHash	86.05%	84.09%	90.38%
Deep Learning	All	86.36%	86.36%	88.89%
Deep Learning	pHash	80.85%	86.36%	88.51%
Deep Learning	pHashMask	80.95%	77.27%	87.47%
Deep Learning	dHash	86.52%	86.36%	86.97%

Table 7.2: Effectiveness of all combinations of classifiers and datasets ranked by AUC

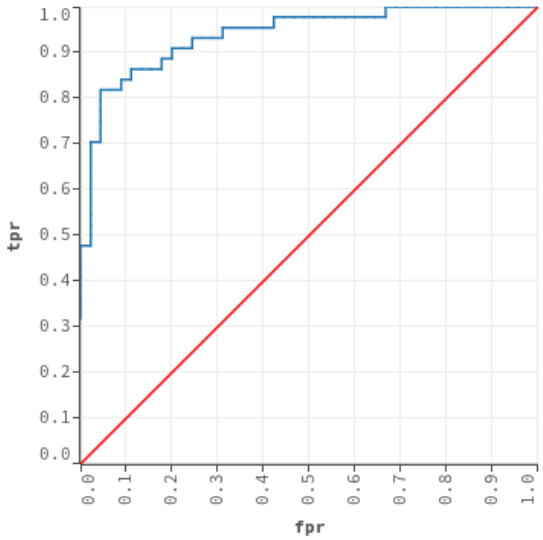

Figure 7.1: ROC curve of best performing Deep Learning classifier with dHashMasked dataset

Actual/Predicted	False	True	Error	Rate
False	43	2	0.0444	2 / 45
True	8	36	0.1818	8 / 44
Total	**51**	**38**	**0.1124**	**10 / 89**

Table 7.3: Confusion Matrix of best performing Deep Learning classifier with `dHashMasked` dataset

7.3 Classifier Effectiveness

On average, the Naive Bayes classifier achieved the highest AUC values, closely followed by the Distributed Random Forest classifier. Its results are very consistent throughout all datasets.

	Deep Learning	DRF	Naive Bayes
AUC	89.16%	91.09%	91.38%
Precision	85.88%	83.13%	83.20%
Recall	83.63%	86.69%	85.45%

Table 7.4: Average effectiveness of classifiers over all datasets

7.4 Feature Evaluation

To evaluate the importance of the feature subsets, models were trained with only one feature set at a time with the Naive Bayes classifier. A model based only on OCR-related features achieved 82.22% precision and the highest AUC value of 87.58%. The model based on redirect performed poorly - This likely is caused by a lack of AAB websites using redirects. The model based on visual similarity features resulted in mediocre 74.51% precision, 86.36% recall and 82.17% AUC.

FeatureSet	Precision	Recall	AUC
OCR	82.22%	84.09%	87.58%
Redirects	49.44%	100.00%	57.47%
VisualSimilarity	74.51%	86.36%	82.17%

Table 7.5: Effectiveness of feature sets with Naive Bayes classifier

7.5 Evaluation with Alexa top 1 Million Websites

To test the model with more representative data, three lists each containing 100 URLs were generated. They are randomly selected samples from the top 1,000, 100,000 and 1,000,000 Alexa pages.

Of those 300 websites 258 (86%) were crawled successfully, the remaining 42 (14%) were unreachable or timed out while loading, so that visual similarity comparison could not be performed. Mughees et al. [51] encountered 558 (0.56%) AAB websites within their dataset of 100.000 websites. therefor it was expected to encounter between zero and one AAB website in each of these datasets.

	1K	100K	1Mio	Total
Candidate URLs	100	100	100	300
Successfully Crawled	89	83	86	258
Positives	14	9	12	35
False positives	11	8	12	31
False negatives	0	0	0	0
True positives	3	1	0	4

Table 7.6: Evaluation of predictions for Alexa top 1.000.000 data samples

The best performing Deep Learning model was used for prediction. In total 35 websites (13,57%) were classified as using AAB scripts. Screenshots of all crawled websites were manually checked for presence of AAB warnings, which revealed 4 true positives and no false negatives.

Page	Alexa Rank	Category	Detection Action	Alexa Sample
redtube.com	206	Pornography	Banner, soft blocking	1K
bild.de	533	News	Redirect, hard blocking	1K
drtuber.com	735	Pornography	Banner, soft blocking	1K
autoplus.fr	26453	News	Pop-up, soft blocking	100K

Table 7.7: True positives

CHAPTER 8

Discussion

In this thesis we discussed the latest developments of the adblocking arms-race. In the State of The Art Section we summarized the technical means available to both sides of the adblocking arms race. Current research shows a shift away from web resource-based adblocking towards active adblocking and stealthy adblocking. In other words adblockers can be put into a position, where they no longer can proactively identify and block HTTP(S) requests to unwanted resources when ads are delivered by the same publisher as the contents of a website. The AAB scripts we encountered in the case study are based on the simple premise that fake bait elements trigger an active adblocker and therefor are not present in the DOM or are hidden by CSS attribute modifications. This can be effective against classical adblockers, but so far all solutions known to the author have been successfully countered by more sophisticated adblockers such as Anti-Adblock Killer, which also inject page-specific JavaScript to disable detection scripts. This challenges the community-based rule generation process of classical adblockers because the page-specific scripts now have more privileges than the previous rules with syntactic limitations. This means that it becomes easier for a malicious attacker to contribute code with undesired side-effects and therefor quality control becomes a greater issue than with classical filter lists.

Even though the user side is in a privileged position because browser extensions can leverage higher privileges than the JavaScript contained in websites, it should not be forgotten that the advertising industry does not only fight adblocking with technical methods, but also uses lobbying and legal actions. The goal doesn't appear to be to completely defeat all adblockers but rather to make it harder for mainstream users to install and configure effective adblockers properly. Nevertheless the economic rules of supply and demand also apply to websites and therefor websites which applied AAB scripts have lost significant numbers of users. If a website's contents are not unique, as is the case with the majority of news websites, annoyed users will fluctuate to competitors, thereby decreasing the website's reach. Only considering direct revenue from ads is a

narrow-minded perspective because adblocker users can still be beneficial to websites by influencing others to visit a website or creating user content such as comments and reviews.

Malvertising is still a valid reason for blocking ads since websites with AAB scripts have fallen victim to malvertising campaigns. By pressuring users to disable their adblockers, publishers knowingly have exposed them to risks without taking responsibility for the incurred damages or providing effective solutions for preventing such incidents in the future.

8.1 Limitations of Visual Anti-Adblock Detection

The results of automated AAB detection with the implemented webcrawler showed that it was capable of identifying true positives. However this was achieved with a rather high false positive rate. After reviewing data and screenshots of the false positives, the following reasons were identified:

The performance of the webcrawler was a limiting factor, because every page needed to be visited at least three times per browser configuration. Due to potentially long loading times and adblock detection being triggered delayed in some instances, it needed to work with conservative timeout limits. Even though crawling processes were run in parallel, due to high memory consumption of Firefox the number of instances was limited to 4 on an Intel Xeon X5460 machine with 8 GB of memory. This in turn limited the sample size and therefor also the amount of data available for training the machine learning models. It is very likely, that the model would yield better results with more training data.

Many of the false positives were websites heavily loaded with dynamic content such as large continuously changing banners filling up large areas of the screen. Some of them also contained videos, which started playback automatically as well as dynamically changing background images.

Figure 8.1: Too much dynamic content to apply visual similarity algorithms. Source: `topix.com`

A known limitation of the perceptional visual similarity algorithms is that they compare the same areas of images and are sensitive to positional changes of page elements. Therefor

large ad banners which move the contents towards the bottom of a page (see Figure 8.2) can cause high difference scores besides the fact that the contents are similar.

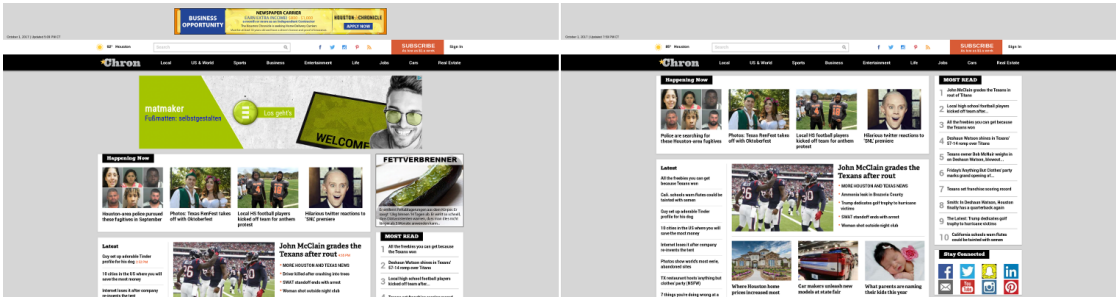

Figure 8.2: Ad banner shifts content downwards. Source: `chron.com`

Another reason for false positives is derived from the inherent problem of long loading times, which also is among the most common motivations for using adblockers. In some cases the pages didn't render correctly due to slow loading of ads, which in turn leads to significant differences in visual appearance and text.

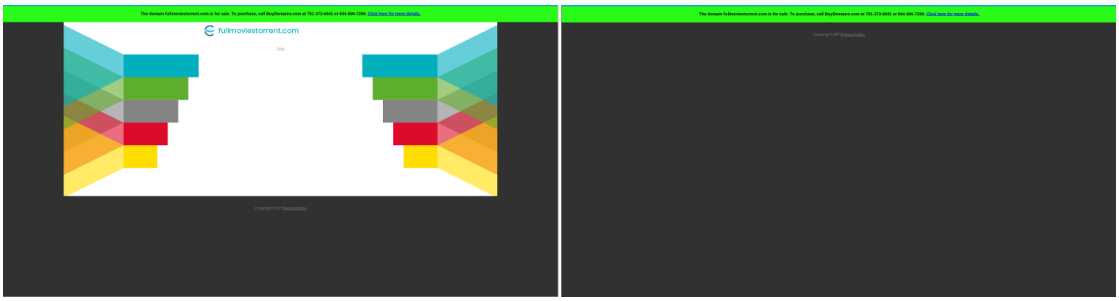

Figure 8.3: Parked domain with only ad as content. Source: `fullmoviestorrent.com`

Towards the lower ranks of the Alexa top 1 million pages there was a large number of parked domains, which didn't have any content besides ads. Due to the ads not being shown anymore with an active adblocker, the visual appearance changed significantly as well as the amount of text on the screen (see Figure 8.3). This clearly demonstrates that the noise of removed ads can also cause false positives.

8.2 Future Work

In the current version the crawler's performance comes close to AAB detection based on analysis of DOM changes. Even though it has a significantly higher false positive rate, visual AB detection has the advantage of being resistant against source code obfuscation and can also detect AAB warnings if they are displayed in the form of an image. For

future research it would be of interest if the combination of both methods could further improve results.

In this thesis only landing pages of websites were analyzed. During the research phase we found that some websites show AAB warnings only on subpages, or limit certain aspects of functionality such as video playback and comments. For a more complete estimate of the number of AAB websites, a systematic approach for testing subpages and individual website features would be required.

Furthermore this work focuses on AAB warnings and content access restrictions. Another aspect that to the knowledge of the author has not yet received attention is silent adblock detection for user tracking. For this purpose as well as for countering AAB scripts a methodology for differentiating between real ads and bait elements in real-time would need to be developed.

CHAPTER 9

Conclusion

In this thesis we researched the current state of the arms race between publishers and adblockers. We found that most anti-adblocking scripts inject bait elements, which are DOM elements or web resources that are blocked by adblockers due to their name, URI or characteristics and in a second step verify the presence of them. Effective measures for circumventing them include manipulation of CSS attributes of anti-adblock warnings, signature-based blocking or manipulation of JavaScript API calls and tampering with browser storage. Whereas detection methods are very similar for most anti-adblocking scripts, they differ in the way they react to active adblockers. Reactions can be categorized as silent, soft and hard blocking. In a case study we found that anti-adblockers can be effective against classical adblockers but are circumvented by more sophisticated solutions. User numbers have significantly declined for websites, which started using them. Among 258 randomly selected websites from the Alexa top 1 million pages we found 4 (1.55%), which hinted their users to disable their adblockers.

Bibliography

[1] Peter Abeln. Ad blocker detection: An overlay modal with randomly generated id. https://github.com/chrisaljoudi/uBlock/issues/1553. Accessed: 2017-06-21.

[2] Digital Advertising Alliance. Adchoices. http://youradchoices.com/. Accessed: 2017-08-03.

[3] Nate Anderson. How a banner ad for H&R block appeared on apple.com — without Apple's OK. http://arstechnica.com/tech-policy/2013/04/how-a-banner-ad-for-hs-ok/. Accessed: 2016-09-28.

[4] Sean Blanchfield. Halloween security breach. https://pagefair.com/blog/2015/halloween-security-breach/?nabe=4982323603046400:1. Accessed: 2016-01-16.

[5] Karl Bode. Mediacom injecting their ads into other websites. http://www.dslreports.com/shownews/112918. Accessed: 2016-09-28.

[6] Karl Bode. Reddit's technology subreddit ponders banning wired & forbes for blocking adblock users. https://www.techdirt.com/articles/20160509/07311734387/reddits-technology-subreddit-ponders-banning-wired-forbes-blocking-adblock-users.shtml. Accessed: 2017-02-21.

[7] Carter Bowles. How the noscript tag impacts seo. https://northcutt.com/how-the-noscript-tag-impacts-seo-hint-be-very-careful/. Accessed: 2016-12-06.

[8] Tim Bray. The javascript object notation (json) data interchange format. 2014.

[9] Johannes Buchner. ImageHash. https://github.com/JohannesBuchner/imagehash. Accessed: 2017-06-12.

[10] Craig Buckler. Average page weight increased another 16 https://www.sitepoint.com/average-page-weight-increased-another-16-2015/. Accessed: 2017-08-05.

[11] Celery. Celery: Distributed task queue. http://www.celeryproject.org/. Accessed: 2017-08-20.

[12] Joseph C Chen. Massive malvertising campaign in us leads to angler exploit kit/bedep. http://blog.trendmicro.com/trendlabs-security-intelligence/malvertising-campaign-in-us-leads-to-angler-exploit-kitbedep/. Accessed: 2016-12-08.

[13] Scott Cunningham. Getting lean with digital ad ux. https://www.iab.com/news/lean/. Accessed: 2017-01-12.

[14] Charlie Curtsinger, Benjamin Livshits, Benjamin G Zorn, and Christian Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *USENIX Security Symposium*, pages 33–48, 2011.

[15] Bundesministerium der Justiz und für Verbraucherschutz. Gesetz über urheberrecht und verwandte schutzrechte (urheberrechtsgesetz) § 95a schutz technischer maßnahmen. http://www.gesetze-im-internet.de/urhg/__95a.html. Accessed: 2016-10-10.

[16] Lewis DVorkin. Inside forbes: More numbers on our ad blocking plan. https://www.forbes.com/sites/lewisdvorkin/2016/02/10/inside-forbes-more-numbers-on-our-ad-blocking-plan-and-whats-coming-next/#332279b96209. Accessed: 2017-02-21.

[17] Easylist. Easylist. https://easylist.to/. Accessed: 2016-08-24.

[18] Easylist. Easylist. https://pagefair.com/blog/2016/how-not-to-deal-with-adblocking-lists-behind-the-scenes-update/. Accessed: 2017-08-07.

[19] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of ACM CCS 2016*, 2016.

[20] Fanboy. Fanboy annoyances list. https://www.fanboy.co.nz/. Accessed: 2016-08-24.

[21] Coalition for Better Ads. The initial better ads standards. https://www.betterads.org/standards/. Accessed: 2017-07-20.

[22] Bundesamt für Sicherheit in der Informationstechnik (Germany). Surfen, aber sicher! https://www.bsi-fuer-buerger.de/SharedDocs/Downloads/DE/BSIFB/Broschueren/Brosch_A6_Surfen_aber_sicher.pdf?__blob=publicationFile. Accessed: 2016-01-15.

[23] Hauke Gierow. Abgemahnter youtuber hat bild.de verklagt. http://www.golem.de/news/streit-um-adblock-video-abgemahnter-youtuber-hat-bild-de-verklagt-1603-119649.html. Accessed: 2016-10-10.

[24] Phillipa Gill, Vijay Erramilli, Augustin Chaintreau, Balachander Krishnamurthy, Konstantina Papagiannaki, and Pablo Rodriguez. Follow the money: understanding economics of online aggregation and advertising. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 141–148. ACM, 2013.

[25] Eyeo GmbH. Allowing acceptable ads in adblock plus. `https://adblockplus.org/en/acceptable-ads`. Accessed: 2016-08-10.

[26] Daniel G Goldstein, R Preston McAfee, and Siddharth Suri. The cost of annoying ads. In *Proceedings of the 22nd international conference on World Wide Web*, pages 459–470. International World Wide Web Conferences Steering Committee, 2013.

[27] Google. Examples of javascript, iframe, and image dart ad tags. `https://support.google.com/dfp_premium/answer/1131983?hl=en`. Accessed: 2016-08-14.

[28] Friedhelm Greis. Bild.de kann adblocker-rate deutlich senken. `https://www.golem.de/news/nach-werbeblockersperre-bild-de-kann-adblocker-rate-deutlich-senken-1511-117293.html`. Accessed: 2017-02-21.

[29] Christian Grothoff, Matthias Wachs, Monika Ermert, and Jacob Appelbaum. Nsa's morecowbell: Knell for dns. 01/2015 2015.

[30] H2O.ai. H2o.ai. `https://www.h2o.ai/h2o/`. Accessed: 2017-08-15.

[31] R. W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29(2):147–160, 1950.

[32] Samuel Hoffstaetter. Python Tesseract. `https://github.com/madmaze/pytesseract`. Accessed: 2017-07-28.

[33] Joel Hruska. Forbes forces readers to turn off ad blockers, promptly serves malware. `http://www.extremetech.com/internet/220696-forbes-forces-readers-to-turn-off-ad-blockers-promptly-serves-malware`. Accessed: 2016-05-15.

[34] Brave Software Inc. Brave. `https://brave.com/`. Accessed: 2017-08-30.

[35] Ben Kneen. How does ad serving work? `http://www.adopsinsider.com/ad-serving/how-does-ad-serving-work/`. Accessed: 2016-08-14.

[36] Barbara Kordy, Piotr Kordy, Sjouke Mauw, and Patrick Schweitzer. *ADTool: Security Analysis with Attack–Defense Trees*, pages 173–176. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[37] Neal Krawetz. Kind of like that. `https://www.hackerfactor.com/blog/index.php?/archives/529-Kind-of-Like-That.html`. Accessed: 2017-06-12.

[38] Neal Krawetz. Looks like it. https://www.hackerfactor.com/blog/index.php?/archives/529-Kind-of-Like-That.html. Accessed: 2017-06-12.

[39] Balachander Krishnamurthy and Craig E Wills. Cat and mouse: content delivery tradeoffs in web access. In *Proceedings of the 15th international conference on World Wide Web*, pages 337–346. ACM, 2006.

[40] Jürgen Kuri. Klage gegen adblock plus: Teilerfolg für springer, niederlage für eyeo bei "acceptable ads". http://www.heise.de/newsticker/meldung/Klage-gegen-Adblock-Plus-Teilerfolg-fuer-Springer-Niederlage-fuer-Eyeo-bei-Acceptable-Ads-3248585.html. Accessed: 2016-08-10.

[41] Van Lam Le, Ian Welch, Xiaoying Gao, and Peter Komisarczuk. Anatomy of drive-by download attack. In *Proceedings of the Eleventh Australasian Information Security Conference - Volume 138*, AISC '13, pages 49–58, Darlinghurst, Australia, Australia, 2013. Australian Computer Society, Inc.

[42] Philippe Le Hégaret, Ray Whitmer, and Lauren Wood. Document object model (dom). *W3C recommendation (January 2005) http://www. w3. org/DOM*, 2002.

[43] Woody Leonhard. Another privacy threat: Dns logging and how to avoid it. http://www.infoworld.com/article/2608352/internet-privacy/another-privacy-threat--dns-logging-and-how-to-avoid-it.html. Accessed: 2016-09-03.

[44] Zhou Li, Kehuan Zhang, Yinglian Xie, Fang Yu, and XiaoFeng Wang. Knowing your enemy: understanding and detecting malicious web advertising. In *Proceedings of the 2012 ACM conference on Computer and Communications Security*, pages 674–686. ACM, 2012.

[45] Jonathan Mayer and Arvind Narayanan. Do not track - universal web tracking opt out. http://donottrack.us. Accessed: 2017-01-07.

[46] Secret Media. A web without advertising - the implications and consequences of adblocking technologies on equal access to free content. http://docplayer.net/2003236-A-web-without-advertising.html. Accessed: 2016-05-10.

[47] Georg Merzdovnik, Markus Huber, Damjan Buhov, Nick Nikiforakis, Sebastian Neuner, Martin Schmiedecker, and Edgar Weippl. Block me if you can: A large-scale study of tracker-blocking tools. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (IEEE EuroS&P)*, 2017.

[48] David Meyer. Here's how the ad-blocking debate just collided with net neutrality in europe. http://fortune.com/2016/02/19/three-network-ad-blocking/. Accessed: 2016-09-28.

[49] Mozilla. Firefox. https://www.mozilla.org/en-US/firefox/. Accessed: 2017-08-28.

[50] Mozilla. Firefox developer tools. https://developer.mozilla.org/son/docs/Tools. Accessed: 2016-12-08.

[51] Muhammad Haris Mughees, Zhiyun Qian, and Zubair Shafiq. Detecting anti ad-blockers in the wild. *Proceedings on Privacy Enhancing Technologies*, 1:16, 2017.

[52] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Security and privacy (SP), 2013 IEEE symposium on*, pages 541–555. IEEE, 2013.

[53] Rishab Nithyanand, Sheharbano Khattak, Mobin Javed, Narseo Vallina-Rodriguez, Marjan Falahrastegar, Julia E Powles, Emiliano De Cristofaro, Hamed Haddadi, and Steven J Murdoch. Ad-blocking and counter blocking: A slice of the arms race. *arXiv preprint arXiv:1605.05077*, 2016.

[54] Body of European Regulators for Electronic Communication. Berec guidelines on the implementation by national regulators of european net neutrality rules. http://berec.europa.eu/eng/document_register/subject_matter/berec/regulatory_best_practices/guidelines/6160-berec-guidelines-on-the-implementation-by-national-regulators-of-european-net-neutrality-rules. Accessed: 2016-09-28.

[55] National Cyber Security Centre Ministry of Security and Justice (Netherlands). Cyber security assessment netherlands 2016: Professional criminals are an ever greater danger to digital security in the netherlands. https://www.ncsc.nl/english/current-topics/Cyber+Security+Assessment+Netherlands/cyber-security-assessment-netherlands-2016.html. Accessed: 2016-12-18.

[56] OpenDNS. A new reason to love OpenDNS: no more ads. https://www.opendns.com/no-more-ads/. Accessed: 2016-09-13.

[57] OpenX. Ad networks vs. ad exchanges: How they stack up. http://openx.com/blog/openx-releases-new-whitepaper-ad-networks-vs-ad-exchanges/. Accessed: 2016-08-12.

[58] Pagefair and Adobe. The 2015 ad blocking report. https://pagefair.com/blog/2015/ad-blocking-report/. Accessed: 2016-05-09.

[59] Pagefair and Adobe. Adblocking goes mobile - pagefair2016 mobile adblocking report, revised november 2016. https://pagefair.com/downloads/2016/05/Adblocking-Goes-Mobile.pdf. Accessed: 2017-02-20.

[60] Pagefair and Adobe. The state of the blocked web - 2017 global adblock report. https://pagefair.com/blog/2017/adblockreport/. Accessed: 2017-02-20.

[61] Wladimir Palant. Javascript deobfuscator. https://palant.de/2009/02/13/javascript-deobfuscator. Accessed: 2016-12-08.

[62] LLC Pi-hole. Pi-hole®: A black hole for internet advertisements. https://pi-hole.net/. Accessed: 2017-09-29.

[63] Inc Pivotal Software. Rabbitmq - messaging that just works. https://www.rabbitmq.com/. Accessed: 2017-08-20.

[64] Flattr Plus. Flattr plus. https://flattrplus.com/. Accessed: 2016-08-10.

[65] Niels Provos, Panayiotis Mavrommatis, Moheeb Abu Rajab, and Fabian Monrose. All your iframes point to us. In *Proceedings of the 17th Conference on Security Symposium*, SS'08, pages 1–15, Berkeley, CA, USA, 2008. USENIX Association.

[66] Enric Pujol, Oliver Hohlfeld, and Anja Feldmann. Annoyed users: Ads and ad-block usage in the wild. In *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*, pages 93–106. ACM, 2015.

[67] Python. Python Software Foundation. https://www.python.org/. Accessed: 2017-08-20.

[68] Sridhar Ramaswamy. Building a better web for everyone. https://www.blog.google/topics/journalism-news/building-better-web-everyone/. Accessed: 2017-08-08.

[69] Raymond.cc. 10 ad blocking extensions tested for best performance. https://www.raymond.cc/blog/10-ad-blocking-extensions-tested-for-best-performance/2/. Accessed: 2016-08-08.

[70] Inc. Red Hat. Fedora. https://getfedora.org/. Accessed: 2017-08-25.

[71] Reek. Anti-Adblock Killer. https://reek.github.io/anti-adblock-killer/. Accessed: 2016-06-05.

[72] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. Detecting and defending against third-party tracking on the web. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 12–12. USENIX Association, 2012.

[73] Selenium. Selenium - web browser automation. http://www.seleniumhq.org/. Accessed: 2017-08-26.

[74] sitexw. Blockadblock. https://github.com/sitexw/BlockAdBlock. Accessed: 2016-12-04.

[75] Ray Smith. An overview of the tesseract ocr engine. In *Document Analysis and Recognition, 2007. ICDAR 2007. Ninth International Conference on*, volume 2, pages 629–633. IEEE, 2007.

[76] StatCounter. Browser market share worldwide. http://gs.statcounter.com/browser-market-share. Accessed: 2017-08-08.

[77] Grant Storey, Dillon Reisman, Jonathan Mayer, and Arvind Narayanan. The future of ad blocking: An analytical framework and new techniques. *arXiv preprint arXiv:1705.08568*, 2017.

[78] Sven T. Jsdetox. http://relentless-coding.org/projects/jsdetox/. Accessed: 2017-07-25.

[79] Ian Thomas. Online ad business 101, part vi – ad exchanges. http://www.liesdamnedlies.com/2008/11/online-ad-business-101-part-vi-ad-exchanges.html. Accessed: 2016-08-13.

[80] Ian Thomas. Online advertising business 101, part ii - how does adserving actually work? http://www.liesdamnedlies.com/2008/06/online-advert-1.html. Accessed: 2016-08-13.

[81] Thomas Unger, Martin Mulazzani, Dominik Fruhwirt, Markus Huber, Sebastian Schrittwieser, and Edgar Weippl. Shpf: Enhancing http (s) session security with browser fingerprinting. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 255–261. IEEE, 2013.

[82] David P. Wiggins. Xvfb - virtual framebuffer X server for X Version 11. https://www.x.org/archive/X11R7.6/doc/man/man1/Xvfb.1.xhtml. Accessed: 2017-08-28.

[83] John Wilander. Intelligent tracking prevention. https://webkit.org/blog/7675/intelligent-tracking-prevention/. Accessed: 2017-06-20.

[84] Wei Xu, Fangfang Zhang, and Sencun Zhu. The power of obfuscation techniques in malicious javascript code: A measurement study. In *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on*, pages 9–16. IEEE, 2012.