Automated data extraction

WHAT YOU SEE
MIGHT NOT BE
WHAT YOU GET

DETECTING WEB BOT DETECTION

Gabry Vlot
STUDENT #851708682

Automated data extraction; what you see might not be what you get

Detecting web-bot detection

Master Thesis

by

G. Vlot

In partial fulfilment for the degree of

Master of Science
in Software Engineering

at the Open University, faculty of Management, Science and Technology
Master Software Engineering

Student number:	851708682		
Course:	AF-SE IM9906		
Date:	July 5th, 2018		
Version:	1.0		
Chairman and supervisor:	Dr. Ir. Hugo Jonker	hugo.jonker@ou.nl	Open University
Second supervisor:	Dr. Greg Alpár	Greg.Alpar@ou.nl	Open University

Table of contents

List of tables

Code snippet index

Illustration index

Diagram index

Abstract

The internet contains a wealth of data that is being used by scientists as foundation for their studies in different research fields.
Scientists are making use of web bots to automatically extract data from web pages on the internet as manually data extraction is infeasible.
Aforementioned implies that a web-bot has become a crucial factor in the execution of a study and therefore it is important that the data extracted by a web-bot is in line with the data presented to a human being when visiting a web page with a regular browser.
Unfortunately, there are indications of limitations in respect to the employment of web bots which can lead to deviations in the extracted data compared to the data of a web page when visited by a human being. A deviation seems to be caused by web-bot detection, the detection of a web-bot by a web site. If studies are not taking into account the deviations in the extracted data between the visit by a web-bot and human being, the results of the study can be misleading or might be even disposable.
To the best of our knowledge there is little to no attention to the risks involved towards the employment of web bots for data extraction in relation to studies. For that reason, we have to doubt the realism of the results described in papers.
This document describes a research towards the adoption of web-bot detection on the internet and relate the observations to the impact on web-bot based studies. This research serves as the first step to investigate the realism of the results of web-bot based studies.
To measure web-bot detection on the internet we propose a methodology to find means to detect web-bot detection, measure the adoption of web-bot detection on the internet and observe the effect of web-bot detection on a web page on the internet.
By conducting a manual observation, we proved the existence of web-bot detection and found means to measure the adoption of web-bot detection on the internet. We found that web-bot detection takes place on the internet based upon a client side approach by observing the properties exposed by browsers. By making use of a browser family fingerprint classification we established a web-bot fingerprint surface. The web-bot fingerprint surface contains specific web-bot browser properties that can be used to detect a web bot.
The adoption of web-bot detection has been measured by making use of a web-bot detection scanner based upon the web-bot fingerprint embodied in detection patterns. The web-bot scanner has been employed in the wild by making use of the Alexa top 1-million-list. The scanner investigated 391517 sites and found sites that contain web-bot detection code inclusions in 12.82% of the sites. All the sites with web-bot detection script inclusions have been classified by the level of web-bot detection. Results indicate that of the sites that contain web-bot inclusions, 91.73% web-bot detection takes place and in 8.25% of the sites browser fingerprinting takes place that can be used to detect a web bot.
Three different types of deviations have been observed on the homepages of 20 sites with web-bot detection inclusions. For each site a screenshot of the homepage has been compared between a visit of a web browser instance and a visit by a human being with a regular browser. The following deviation types have been distinguished: *blocked response, blocked web page content* and *deviation in the content of the web page*. In respect to web based studies in the *deviation in the content of the web page* is the most harmful deviation type because this deviation is hard to detect.
We argue that web-bot detection is a real threat for web-bot based studies although this research is not able to *prove* the existence of web-bot detection for all sites with web-bot detection script inclusions. The adoption rates indicate a high coverage of web sites that are able to detect a web bot. Manual observation of different types of deviations indicate that the results of web based studies can be thwarted by web-bot detection.
Future work that combines web-bot detection with deviation detection is necessary to prove the existence of web-bot detection for all the observed sites with web-bot detection script inclusions.

1 Introduction

Scientists are making use of *web bots* in their studies to automatically extract data from web pages. Do these *web-bot based studies* get the same results compared to a human being that visits a web site by making use of a regular browser?

A web-bot in the context of the project described is an automated agent that uses specific software (for instance PhantomJS[1] or Selenium Webdriver[2]) to extract data from web pages.
Employment of web bots are applied in different research fields like for instance privacy and social media.
To give an example: in the field of privacy it is the work of Nikiforakis et al. that provides insights in the ecosystem of web-based device fingerprinting [NKJ+13]. In their work they used a web-bot to extract the content of web pages to observe code inclusions related to fingerprinting.
An example in the field of social media is the work of Dai et al. that researched the characterization of LinkedIn[3] profiles by making use of a web-bot [DNV15]. The web-bot in the work of Dai et al. was used to extract profiles from the LinkedIn web site to provide insights in hidden relationships between the profiles.

Despite the fact that different web-bot based studies serve a different purpose, they have one thing in common: they all heavily depend on data that has been extracted by making using of a web bot. A web-bot has become a crucial factor in the execution of a study. It is therefore extremely important that the extracted data by a web-bot is reliable and is in line with the data presented to a human being when visiting a web page with a regular browser.

Unfortunately, there are some indications of limitations in respect to the employment of web bots as described in the work of Englehardt et al.
Englehardt et al. measured the adoption of online tracking by making use of web bots for the extraction of tracking data from web pages [EN16]. During their study Englehardt et al. observed that advertisements are not being served to a web-bot that makes use of PhantomJS. Englehardt et al. described:

> […] *Upon further examination, one major reason for this is that many sites don't serve ads to PhantomJS.*

Limitations in the employment of a web-bot has also been observed during an initial case study towards price discrimination. During this case study a web-bot has been employed for extracting flight information from web sites of airline companies.
Particular airline companies blocked a visit by a web-bot on their sites by making use of a CAPTCHA[4] contrary to a page visit by a human being.
A CAPTCHA is a test that replaces the main contents of the web page that requires interaction of a user to determine whether or not the user is a human being. A CAPTCHA is hard to resolve for a web-bot as it is designed for human interaction.

The aforementioned limitations causing *deviations* in the extracted data by a web-bot (that represents the content of a webpage) in respect to the data of a web page visit by a human being.
Deviations seem to occur because the web-bot can be detected by a web site.
In other words; when a web site is able to detect a visit of web site by a web-bot (*web-bot detection*), it is able to return data that is specific to a *visitor class*: web-bot or a human being.

[1] http://phantomjs.org/
[2] https://www.seleniumhq.org/projects/webdriver/
[3] https://www.linkedin.com/
[4] https://en.wikipedia.org/wiki/CAPTCHA

Web-bot detection can have a tremendous impact on the outcome of a study.
Consider the work of Yajin et al. [YZW+12] where Android Markets are being explored for malware by making use of a web bot.
Imagine that Google is able to detect the prior-mentioned web-bot that being used to extract the data. Google might provide different data (different apps) depending on the visitor class and hide apps from suspicious providers in case of a web bot.
What does this mean for results of the study?
Aforementioned is illustrated in Figure 1-1 that describes web-bot detection based upon a web page visit by a web-bot on the left side and web page visit by a human being on the right side. *Web site A* represents an Android store and the documents representing the available apps.
The blue document represents the data with only the apps from being software providers while the green document represents all the apps available in the Android store (which also includes the apps from suspicious software providers).

Another example is a (fictional) study that researches the competences between students from different universities by making use of a web bot. The web-bot extracts student profiles from the university web sites including their course degrees.
The result of the gathered data by the web-bot contains for one particular university only students with degrees above the national average. In contrast; a manually visit of the university web site, lists all the students of the university including the students with degrees below the national average.
Will the results of this study be realistic?
Figure 1-1 can also be used to illustrate the aforementioned situation. In this case *Web site A* represent the university web site and the blue document represents the data with only the students a degree above the national average and the green document represents only all the students of the university.

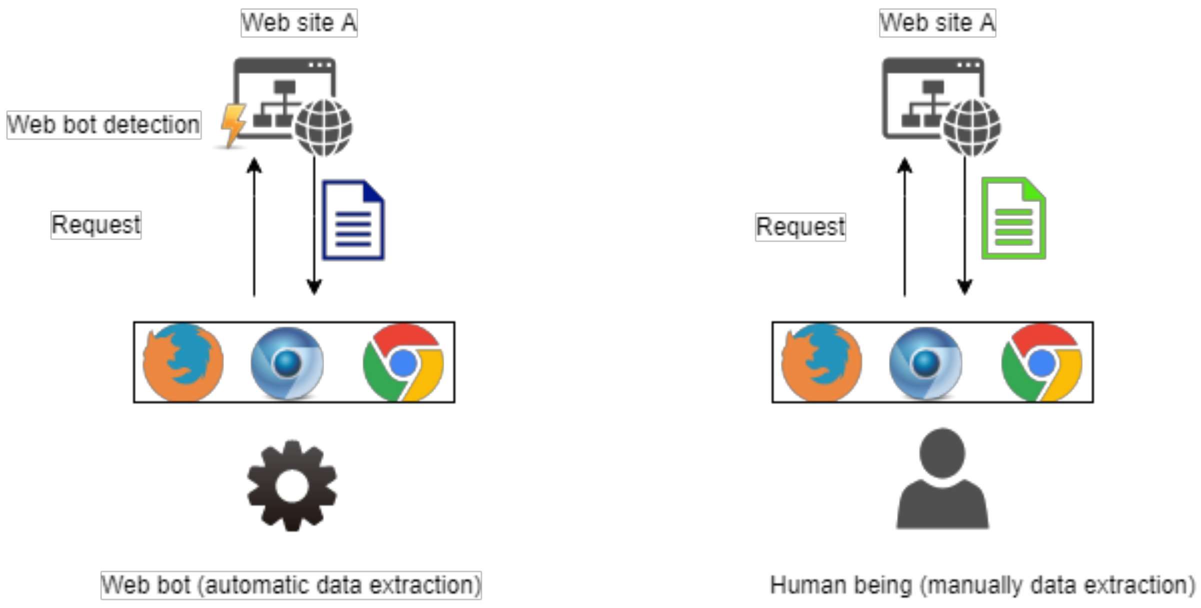

Web site A
Web site A
Web bot detection
Request
Request
Web bot (automatic data extraction)
Human being (manually data extraction)

Figure 1-1: Data extraction; web-bot versus human being

The aforementioned examples described that if a web-bot can be detected by a web site, deviations in the extracted data might occur which can negatively influence the results of a study.
If studies are not taking into account the deviations in the extracted data between the visit by a web-bot and human being, the results of the study can be misleading or might be even disposable

To the best of our knowledge there is little to no attention to the risks involved towards the employment of web bots for data extraction in relation to studies.
For that reason, we have to doubt the realism of the results obtained by a study leading to the question: are the results described in research papers of web-bot based studies realistic?

To be able to answer the aforementioned question it is necessary to have more knowledge about the current state of web-bot detection on the internet.
After acquiring this knowledge, web-bot detection can be related to past studies and express the realism of the results described in papers.
In other words; the above-mentioned question has too much research topics to answer within one research project. It needs prior work in which a study towards web-bot detection is the first step and the main focus of the study described which leads the main research question:

To what extent is web-bot detection adopted on the internet and how does it relate to studies that are using web bots for automated data extraction?

This main research question can be decomposed into the following sub research questions:

RQ1: How to proof and measure the existence of web-bot detection?
There are indications that web-bot detection takes place on the internet but there is no well-founded prove of its existence. The answer to this question provides us with the means to measure web-bot detection on the internet (RQ2).

RQ2**: **To what extent is web-bot detection adopted by web sites on the internet?
The answer to this question will provide us with adoption rates in relation to web sites that express the relevance of web-bot detection on the internet. The adoption rates can be used to express the relevance of web-bot detection on the internet in relation to web based studies.

RQ3: Can we distinguish different types of deviations based upon web-bot detection?
By answering this question, we gain more insights in the seriousness of deviations caused by web-bot detection in relation to web-bot based studies. It expresses the relation between web-bot detection and its impact on studies.
Depending on the deviation type, the level of impact differs. For instance, when a deviation concerns blocking behaviour, this could be easily noticed during the study and the impact could be mitigated. On the other hand, subtle deviations could slip through.

RQ4: How likely is it for web-bot based studies to get detected by web-bot detection?
This research question provides insight in the risk of being detected related to used data extraction techniques and approaches.
By answering this question, we express the risk of being detected that indicates (together with RQ2) the relevance of web-bot detection on the internet in the context of web-bot based studies.

The proposed study answers the aforementioned questions by providing the following main contributions:

- We prove that web-bot detection takes place by analysing a client side web-bot detection implementation.
- We substantiate that web-bot detection can be measured based upon browser properties by examination of the innerworkings of a web-bot detection mechanism.
- We developed a classification method by utilizing a web site that can be used to distinguish web-bot driven browsers from human driven browsers based on the properties exposed by a browser.
- We developed a web detection scanner that is able to measure the adoption of web-bot detection on the internet based on web-bot fingerprinting patterns.
- We present adoption rates that express the coverage of web-bot detection on the internet.
- We prove that web-bot detection can lead to different types of deviations by making use of a manual observation.
- We argue that web-bot detection is a real threat for the results of studies that employing web bots to extract data from web pages on the internet.

Thesis overview
This Thesis is organized as follows:
Chapter 2 provides the necessary background information towards web-bot detection, different types of web bots, data extraction tactics and provides in examples of web-bot based studies.
Chapter 3 describes related work to emphasize the fact that the research field related to browser based web-bot detection is untouched and that past web-bot based studies lack the attention towards web-bot detection.
The methodology used during the research will be described and explained in chapter 4.
Chapter 5 describes the first step taken in the research which is an analysis of a client web-bot detection implementation. The analysis proves the existence of web-bot detection and determines its usage for measuring web-bot detection on the internet.
Chapter 6 describes a family fingerprint classification that is being used to automatically detect web-bot detection based upon specific browser properties gathered by making use of the browser families classification.
Chapter 7 describes the approach to obtain web-bot browser properties and determining of the web-bot finger print surface.
Chapter 8 describes the approach and results of measuring the adoption of web-bot detection in the wild by making use of a web-bot detection scanner.
Chapter 10 concludes this Thesis by describing our conclusions based upon the answers to the research questions, introduction of discussions and an outlook to future work.

2 Background

This chapter describes background information concerning:

- The different ways and commercial implementations of web-bot detection (cf. paragraph 2.1).
- The existence of different types of web bots (cf. paragraph 2.2).
- The existence of different types of data extraction tactics (cf. paragraph 2.3).
- The variability of studies that are employing automated data extraction (cf. paragraph 2.4).

The information serves as context for the proposed study to understand the choices we made in respect to the methodology (cf. chapter 4) and setting the domain of the study.

2.1 Web-bot detection

In the context of this study, web-bot detection refers to the detection of automated software for the purpose of extracting data from the content of a web page.
A web-bot is not only being used for benign purposes, like for instance for gaining knowledge in context of science but also for malicious reasons for example; steeling copyrighted content / intellectual property or the execution of a *ddos attack*[5].
Web bots are popular for the employment of criminal activities because they are easy to scale and mimic human behaviour which makes them harder to track.

Web site owners are protecting themselves from web bots by employing web-bot detection by observing the visitor, looking for signs of the behaviour of a web bot.
Also, commercial cybersecurity companies like for instance WhiteOps[6] and Distil Networks[7] offer web-bot detection as a service to their customers (cf. paragraph 2.1.3).

2.1.1 Web-bot detection on different implementation levels

A web-bot can be detected on different levels of the OSI-model[8] which standardizes layered processing of communication between computers.
Table 2.1 describes the relevant layers of the OSI-model in which a web-bot can be detected.

OSI-Layer	Web-bot detection
Network (Layer 3)	A web-bot can be detected on the *Network layer* if excessive network traffic has been monitored from repeating identical IP addresses (or pool of IP addresses)
Transport (Layer 4)	In the *Transport Layer* web bots can also be detected based upon TCP traffic on the network. For instance, traffic generation based on click rates (For instance, sending x requests per second). Exceeding incoming TCP packages makes the web-bot standout above all other network interaction. Behavioural patterns in the traffic, like a page visit each Monday at four in the afternoon, could make a visitor suspicious. This classification has also been described in the Distil whitepaper [Dis16] that concludes; *“many bots nowadays are mimicking human behaviour; bad bots begin to work from 9 am till 5 pm*” [Dis16].

[5] https://en.wikipedia.org/wiki/Denial-of-service_attack
[6] https://www.whiteops.com/
[7] https://www.distilnetworks.com/
[8] https://en.wikipedia.org/wiki/OSI_model

	Not only the timeslot can be of importance but also the repeating sequence of user interaction within a browser. For instance, a click on the same button on the left top corner followed by the same link. Behavioural patterns can be tracked by monitoring network traffic within the transport layer.
Application (Layer 7)	The HTTP[9] protocol in the application layer, which is the protocol for the communication between web client (browser) and web server, can contain information that could reveal the presence of a web bot. Concerning the information specified in the HTTP header, we can distinguish two categories of information: 1. Web-bot HTTP characteristics 2. Web-bot detection context The first one exposes web-bot specific communication properties like the value of the *userAgent*[10] string in the HTTP header or missing or additional HTTP headers.
User (Layer 8)	On layer 8 additional web-bot specific information can be collected client side by making use of client scripting. Typically, those client-side properties are being communicated to the server, by the HTTP protocol (layer 7). The information provided which involves the context for the detection of the web-bot can contain the *characteristics of a web-bot* or *behavioural information:* • **Web-bot characteristics** The properties of the tooling (library/framework) that has been used for automated data extraction. • **Behavioural information** Information that reveals a web-bot by its client-side behaviour. An example of behavioural information is the filled in field values of invisible fields send to the web server by the HTTP protocol. Typically, those fields are added to a web page to detect a web-bot (a honeypot[11]). A human being is not able to fill in those fields as they are invisible. If a webserver notices that an invisible field value has filled in, it knows that it deals with a web bot.

Table 2.1: Web-bot detection per OSI-layer

[9] https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol
[10] https://www.w3schools.com/jsref/prop_nav_useragent.asp
[11] https://en.wikipedia.org/wiki/Honeypot_(computing)

2.1.2 Page deviations caused by web-bot detection

The detection of a web-bot can take place both on server and client side, this in contrast to the determination of the response of the server back to the client which (obviously) takes place server side. The server is always in charge of what will be send back to the client (browser) which can be influenced by the detection of a web-bot which can lead to a deviation on a web page. A *deviation* is an observable difference on a web page caused by the employment of a web-bot in comparison to the web page visited by a human being.

A deviation on a web page can exists in many forms. On the highest level we can distinguish between a deviation in the visual representation and a structural deviation.

- **Visual representation**
  The visual representation, represents the experience of an end user of a webpage in a web browser. The visual representation includes both graphical and audio representation (technically an audio-visual representation).
- **Structural representation**
  The structural representation, represent the DOM tree, parsed from a HTML file. The DOM tree represents the structure of the webpage in the form of a document. The tags defined in the HTML file are structured in the DOM tree as elements. Inline specifications like style and scripting are defined but not applied.

Figure 2-1 describes different server responses that can lead to a deviation on a web page in the visual and / or structural representation.

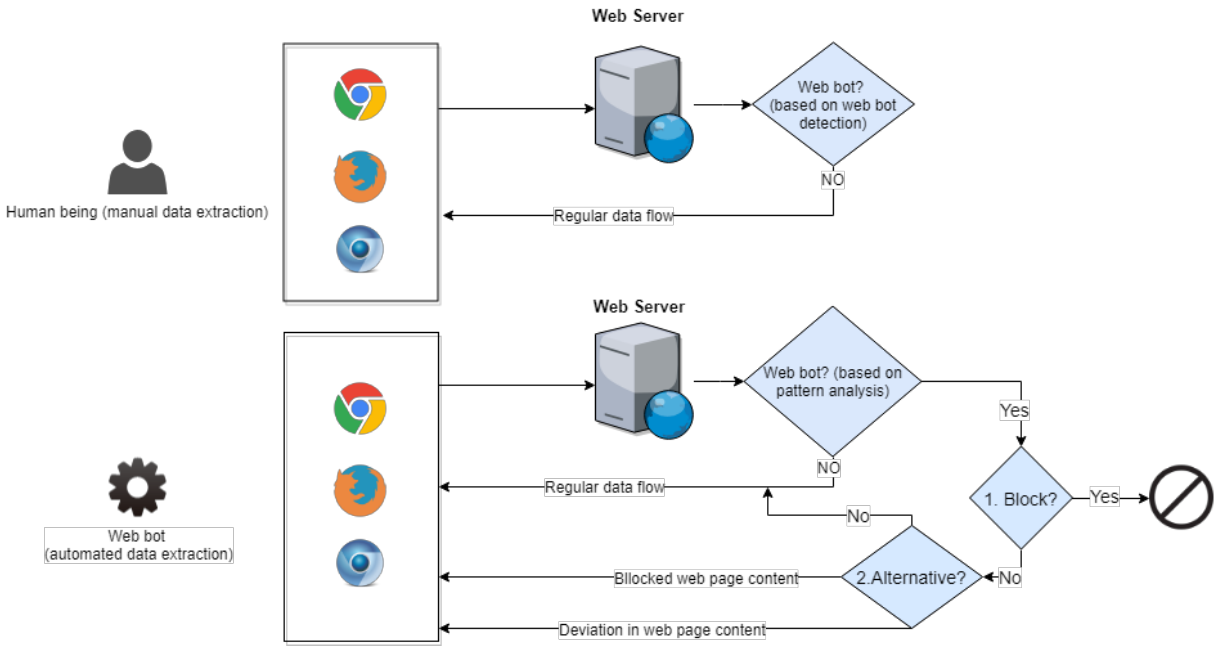

Web Server
Web bot? (based on web bot detection)
NO
Human being (manual data extraction)
Regular data flow
Web Server
Web bot? (based on pattern analysis)
Yes
NO
Regular data flow
1. Block?
Yes
No
Web bot (automated data extraction)
Bllocked web page content
2.Alternative?
No
Deviation in web page content

Figure 2-1: Possible server responses based on web-bot detection

At the top of Figure 2-1 a web page visit by a human being is displayed and in the bottom a web page visit by a web-bot is being displayed.

The latter illustrates that when a web-bot has been detected the *server host* (the owner of the web site or the owner of a library included on the web site) can choose to block the client or to provide alternative feedback leading both to deviations on a web page.

1. **Block**
   In this case the server host chooses to send no response to the client at all. This might cause that the contents of a web page will not change this in contrast to visit to the web page by a human being.
2. **Alternative**
   a. **Blocked web page content**
      The content of the web page has been blocked; browsing the web page (or extracting data from it) is not possible. Blocking can be embodied on a site by replacing the main content with a CAPTCHA.
      Blocking of the content of a web page is usually obvious to the end user (human being or web bot) of the web page.
   b. **Deviation in web page content**
      This is a response that a human being would **not** experience by making use of a regular browser. It is also the most dangerous and hardest deviation to grasp. For instance, imagine a situation where only one word on a page has been changed. In respect to studies this response could harm the outcome of a study the most because it is hard to detect.

If the server host does not choose to block the response or to provide an alternative response the result returned by the webserver is the same in comparison to a visit of a human being with a regular browser (a web page with the same look and feel).

2.1.3 Web-bot detection by commercial companies

This paragraph describes two white papers originating from commercial cybersecurity companies regarding the adoption of web-bot detection. Both whitepapers illustrating the motivation for the need for web-bot detection (from the perspective of the companies described) and the actuality of web-bot detection nowadays.

The detection of "Methbot"
The WhiteOps company released a whitepaper in relation to the existence of a web-bot dubbed "Methbot" [Whi16]. Methbot attacks the digital advertising market by generating fraudulent revenues. The work of White Ops describes that it can identify Methbot based upon a unique signature. The paper reveals very little technical details about Methbot or its detection, although it describes that Methbot tries to appears as a human being by:
- Stuffing crafted cookies into fake web sessions
- Fake cursor movements and clicks
- Forge social network login

The bot runs under Node.js[12] and uses several open source libraries. White Ops has exposed IP addresses and a falsified domain and URL lists to defeat this web bot.

Classification of detected behaviour
The work of Distil Networks discusses the rise of so called Advanced Persistent Bots (APB) [Dis16]. An advanced persistent bot is a bot that owns several advanced capabilities. It includes mimicking human behaviour, loading JavaScript and external resources, cookie support, browser automation, and spoofing IP addresses and user agents.
Distil networks analysed an increasing of human traffic and mentioned the disguise of a bad bot by an APB as one explanation.
The work also presents several metric details like the most used user agent (Firefox), top origination countries, top originators and the like. The work concludes with the observation that many bots nowadays are mimicking human behaviour; bad bots begin to work from 9 am till 5 pm and bots rotate user agents and IP addresses.
The study shows that bots are being detected with a lot of details with them. It also illustrates that mimic human behaviour is a hard thing to do for a web bot.

[12] https://nodejs.org

Both studies reveal that the phenomenon 'web-bot detection' exists and occurs in practice which serves as the motivation for the study proposed.
The data extracted by making use of web bots can contain deviations as cause of web-bot detection. Unfortunately, no prove or detailed technical information is available that can be used for measuring the presence of it on the internet which has influenced the methodology of our study (cf. chapter 3).

2.2 Types of web bots

We can distinguish different types of web bots based upon their data extraction strategy (cf. appendix A) as described in Table 2.2.
With *data extraction strategy* we mean: the strategy of targeting a website for data extraction (i.e. Browser or HTTP request).

Type	Example	Strategy	Description*
HTTP based	WGET	HTTP requests	This web-bot deals with HTTP requests on a lower level. It basically sends and receives HTTP request. No dynamic data can be executed.
Headless browser	PhantomJS	Browser	Stripped down browser without graphical interface.
Headful browser	Selenium + webdriver	Browser	(Full-fledged)-automated scriptable browser

Table 2.2: Different types of web bots based upon data extraction strategy

* See appendix A: Taxonomy of automatic data extraction methods for a more detailed description.

In the context of the study, the HTTP based web-bot is irrelevant. Reason for it is that the server host does not need to have any web-bot detection mechanisms in place to detect this type of web bot.
For a server host it is sufficient to only implement dynamic content to detect a HTTP based web bot. When the dynamic code does not get executed the server host knows that the client (human being or web bot) is not making use of a (regular) browser. Detection of this detection mechanism is useless as it would label all web sites making use of scripting as a web-bot which would result over the 90 per cent coverage on the internet[13].

Headless and headful browsers

The most relevant and popular web-bot in the light of the proposed study is the *browser* type. A browser based web-bot is basically a scriptable browser often supported by a framework that offers the necessary tools for extracting data from webpages.
A headless browser is a specifically developed program that mimics an actual browser without displaying.
A headful browser is an automated full-fledged browser by making use of a driver that is able to send instructions to the browser.

There can be an important difference between a headless and headful browser as also described in the work of Englehardt et al. referring to PhantomJS (a headless browser) in comparison to a headful browser:

> [...] *PhantomJS loads about 30% fewer HTML files, and about 50% fewer resources with plain text and stream content types*. [EN16]

[13] https://w3techs.com/technologies/details/cp-javascript/all/all

In general; a headless browser is more efficient with system resources which makes a headless browser more scalable. In contrast; a headful browser is able to load more web site resources (see citation of Englehardt et al. above) because it is a full-fledged browser instance.
A browser-based web-bot comes with the main advantage that it mimics the behaviour of a human being that is using a browser. Mimicking a human being is an advantage concerning data extraction because:

1. It extracts the same data like a regular browser user will see. For a scientist it is important that the data extracted reflects the reality.
2. It makes it harder for the target website to distinguish a normal user from a web bot.

2.3 Data extraction tactics

Studies are using different data extraction tactics (cf. paragraph 2.4). By a tactic we refer to the relation between web-bot and web site as described in the work of Schrenk [Sch12]. Schrenk categorizes four tactics (referred to by Schrenk *as environment scenarios*) as illustrated in Figure 2-2:

- **One-to-many (A)**
  Is one web-bot that runs many times to gather information from multiple web pages.
- **One-to-one (B)**
  Is one web-bot that gathers information from a single target and repeat this process many times over an extended period of time.
- **Many-to-many (C)**
  Multiple web bots gathering information from multiple web pages.
- **Many-to-one (D)**
  Multiple web bots gathering information from a single web page.

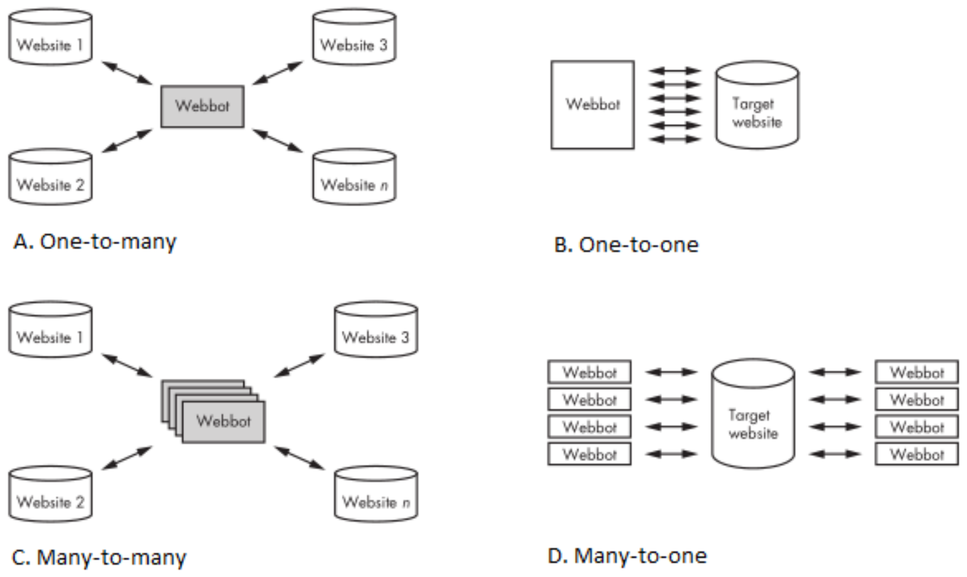


Figure 2-2: Web-bot employment tactics [Sch12]

In the context of the study the data extraction tactic is important as it can be of influence in relation to the detection of a web bot.
A web-bot employed based on a one-to-one tactic will more detectable because it targets the same web page using the same web bot. Based upon traffic or unique signature the web-bot is more likely to be detected than for instance a one to many tactic that will target a specific web page less frequent.

2.4 Studies based on automated data extraction

This paragraph describes studies that are employing automated data extraction for gathering data that serves as foundation for their studies. Each study has been categorized in one of the following categories; *privacy, security, social media* or *networking.*
Per research category we give insights in the techniques and approaches applied.

Privacy
The work of Nikiforakis et al. describes a research towards the ecosystem of web-based device fingerprinting [NKJ+13]. In their work, Nikiforakis et al. crawled popular websites looking for code inclusions originating from three commercial companies for fingerprinting purposes.
The used web-bot crawled up to 20 pages for each of the Alexa top 10.000 sites. Nikiforakis et al. investigated the extracted data like browser plugins (Flash) and web pages for script inclusions (JavaScript / ActionScript), bypassing of HTTP proxies and iframes developed for fingerprinting purposes.
The results are categorized based on features like *Flash enabled*, *screen resolution* and the like. Their work also shows the adoption of device fingerprinting categorized by domain categorization of a popular anti-virus vendor.
The work of Englehardt et al. presents a study of a measurement of online tracking by crawling 1 million websites (visiting only the homepages) by making use of the framework: OpenWPM [EN16]. OpenWPM is an open-source web privacy measurement framework which implements two categories of web bots for extracting data from the internet; 1 lightweight browsers (like PhantomJS) and 2 Full-fledged browsers (cf. appendix A): like Chrome and Firefox.
By making use of customer browsers all techniques like HTML5 storage options and Flash are supported.
There are similarities with the work of Nikiforakis et al. in that it discusses fingerprinting technologies and techniques and the adoption of them in the wild by analysing the data. The work of Englehardt et al. is different by the fact that it discusses the categorisation between first and third party page visits in respect to tracking context.
The work of Acar et al. sheds light on current device printing practices by making use of *FPDetective* [AJN+13]. FPDetective is a framework for identifying and analysing web-based device fingerprinting. The framework uses two lightweight automated browsers for extracting webpages and browser plugin objects from the top million Alexa websites. For the detection of Flash-Based fingerprinting all browser traffic is directed through an intercepting proxy. Ascar et al. analysed extracted scripts and decompiled Flash source code for fingerprinting behaviour.
Current study by Krumnow[14] investigates price discrimination on the internet. Krumnow makes use of a data extraction framework that uses a headless browser (PhantomJS[15]) for extracting data from web pages.
Krumnow targets specific airline sites extracting price information regarding specific flights.

Security
The work of Rieck et al. presents Cujo, a system for automatic detection and prevention of drive-by-download attacks [RKD10]. Cujo inspects web pages and blocks delivery of malicious JavaScript code. Rieck et al. used two different data sets, containing web site URLs, to study the detection and runtime performance in detail:

- *Data sets for benign websites*
  The Alexa top 2000,000 most visited web pages and *Surfing* comprises 20,283 URLs of web pages visited during usual web surfing.
- *Attack data sets that have been adopted from related work*

The URLs in the datasets are being used for extracting data from websites. The HTML and XML documents of the extracted data are inspected for the occurrences of JavaScript. All embedded code

[14] Benjamin Krumnow , Price Discrimination Project http://www.open.ou.nl/hjo/supervision.htm
[15] http://phantomjs.org

blocks like *script* and event handlers like *onmouseover* are extracted from the documents and merged for further static and dynamic analyses.
The work of Rydstedt et al. surveyed the frame busting practised of the Alexa top 500 websites [RBB+10]. Rydstedt et al. used a Java-based emulator HTMLUnit[16] for extracting JavaScript code used in the web site contents. The JavaScript code is being analysed for locating and extracting frame busting code. HTMLUnit is being used as headless emulator for debugging JavaScript to be able to be able to dynamically frame pages and break at the actual script used for frame busting.
The work of Zhou et al. presents a study concerning the detection of malicious applications on popular Android Markets [YZW+12]. In their research they use a crawler build in an android app 'DroidRanger' to explore the apps published in 5 different Android markets. The automatically extracted data includes Android packages that are being analysed for permissions behaviour and dynamic loading.

Social media
The work of Dai et al. presenting a study towards the characterization of LinkedIn[17] profiles. By employing a web-bot for extracting data from specific Linkedin profiles, academic backgrounds are studied and a classification of education levels has been constructed [DNV15]. Dai et al. scraped specific data instead of crawling to get the maximum number of public profiles in a minimum amount of time. The Scrapy[18] framework was used to extract data from specific LinkedIn profiles and analysed by using regular expressions corresponding to XPath for discovering hidden relationships in HTML files.
The work of Catanese et al. describes the collection and analysis of massive data describing the connections between participants to social network data collection [CMF+11]. They define and evaluate approaches to social network data collection in practise against the Facebook[19] website. Catanese et al. used two different web bots to extract friend lists. The web bots are extracting a friendlist graph by sending HTTP requests based on a queue that contains a list of users that will be visited.

Networking
The work of Ager et al. studies the DNS deployment in commercial Internet Service Providers (ISPs) and compare the results to widely used third-party DNS resolvers (GoogleDNS and OpenDNS) [AMS+10].
In respect to their study they are interested in the DNS performance perceived by end-users and therefore make efforts to collect data directly from users' computers.
They wrote code that performs DNS queries for more than 10,000 hosts relying on different DNS resolvers. Their study crawled the top of the Alexa 1,000, 000 sites for acquiring bias between response times. The content of the extracted data is analysed for resolving host names. Ager et al. are using Wget[20] for downloading the content of these hosts.

Relational
All of the aforementioned studies are extracting data from the internet by using a web bot. Only in the work of Englehardt et al. a deviation based on the employment of a web-bot is mentioned.
Although it has been mentioned, no possible causes or consequences are described.
This paragraph illustrates the topicality of automated data extraction performed by web-bot and that there is little to no attention for web-bot detection in relation to the taken approach during a study.

On top of that; these studies also illustrate that there are various techniques and tactics for data extraction available.

[16] http://htmlunit.sourceforge.net
[17] http://linkedin.com
[18] http://scrapy.org
[19] http://facebook.com
[20] https://www.gnu.org/software/wget/

- *Extraction techniques*
  For instance, a browser based approach or a HTTP request approach.
- *Extraction tactics*
  The differences in applied tactics is strongly related to the goal of the particular study. The work of Krumnow[21] extracts data from specific sites and repeats this process many times (one to one environment [Sch12]) identifying something unknown (black box). The other described studies are extracting data in the wild, targeting information from a variety of web sites (one to many or even many to many [Sch12] while knowing what to look for.

This knowledge (data extraction techniques and tactics) is valuable in respect to:

- Expressing of the relation between web-bot detection and studies applying different techniques.
- Choosing the right technique for the proposed methodology (cf. chapter 3).

[21] Benjamin Krumnow , Price Discrimination Project http://www.open.ou.nl/hjo/supervision.htm

3 Related work

This chapter describes related work of the study proposed which can be divided in the categories: *bot detection* and *fingerprinting*.

Bot detection

Related work in respect to the proposed study can be distinguished in bot and web-bot detection. Bot detection is related to the identification of bots in relation to the internet based on traffic analysis. Web-bot detection concerns the detection of a bot toward the web related to the HTTP protocol based on server logs.

Bot detection is rich research field concerning the detection of web bots / botnets based on traffic analysis.
For instance, the work of Hsu et al. is focused on the detection of web bots based upon the fast-flux network architecture[22] [HHC10]. Hsu et al. proposing a schema which is being used to detect a web-bot based upon delays in IP-traffic and the origin of DNS records.
Similar; Liu et al. propose 'BotTracer' that is able to detect bot malware containing the phases [LCY+08]:

- Automatically startup of bot without any user actions.
- Communicating client command to server
- Attack performed by web bot.

BotTracer is able to detect malware activities by web-bot by monitoring machine processes and traffic transferring from client to server.

With regard to web-bot detection based on server logs: Tan et al. propose a method that uses navigational patterns in the click-stream data to determine if it is due to a robot [TK02]. Tan et al. observe different navigational patterns in web server logs.
The work of Yu et al. describes a research of search bot traffic from search engine query logs [YXK10]. Yu et al. present SBotMiner that is able to identify stealthy search bot traffic. SBotMiner gathers groups of users who share at least one identical query and click. Furthermore, SBotMiner examines the aggregated properties of their search activities.

Although the aforementioned studies are related by the fact that bot detection takes places, their approach and goal is different in respect to the proposed study.
In general, we came across bot detection studies that focus on detecting malicious activities by analysing network activities.
In contrast; the focus of our study is on the quality of extracted data by web bots which can be employed for malicious purposes but also for benign purposes like for instance acquiring scientific knowledge.
On top of that, our study differs from related work because the study described researches a web-bot browser approach which has totally has been executed from a client perspective.
We believe that a web-bot browser based detection approach is more in line with the evolution of web techniques that can easily mimic the behaviour of a human being using a browser. This makes it hard to detect a web-bot based on network activities (traffic and / or server logging).
To the best of our knowledge this is the first study towards measuring the adoption of web-bot detection on the internet using a browser based approach.

Fingerprinting

The work of Torres et al. where a prototype tool *FP-Block* is related to the proposed study based upon the usage of browser fingerprinting [TJM14]. The fingerprint of a browser is a set of characteristics exposed by a browser that can be used to uniquely identify an instance of a web browser.

[22] https://en.wikipedia.org/wiki/Fast_flux

FP-Block counters cross-domain fingerprint-based tracking in a way that it does not have impact on the embedded context of a web page. Torres et al. used the concept *web identity* which is a set of browser fingerprinting characteristic to counter the blocking and still maintain the functionality provided by the embedded plugins.
The work of Torres et al. established a *fingerprint surface* from which the *web identity* could be generated. The *fingerprint surface* represents the distinguishing browser characteristic that can be used to identify a web browser instance. The establishing of the *fingerprint surface* and the use of it, are both closely related to proposed study where a *fingerprint surface* is create also based upon browser characteristics to identify web-bot browser instances. The work of Torres et al. has directly been used to determine relevant browser properties (cf. chapter 7).
The work of Mudialba et al. is also related to our study as it examines the behaviour of a web page depending the ability of tracking based on fingerprinting [MNM17]. Mudialba et al. combined two tools respectively: *Ad Fisher* [SIM17] and *FPBlock* [TJM14]. They found statistically significant evidence that fingerprinting occurs on certain web services. *Ad fisher* is a tool that investigates the segmentation of the ads displayed by Google to web pages of third parties. Mudialba et al. found that there is indeed a difference between ads provided to browser instances when blocking tracking by modifying its fingerprint and without blocking tracking. Their study is directly related to the proposed study because it investigates the effect of the browser fingerprint on the content of a webpage. In our study we examine indirect the effect of a browser fingerprint on the content of a web page in combination with the presence of web-bot detection of the web site.

4 Methodology

In this chapter we propose a methodology, as illustrated as a framework in Figure 4-1, to investigate the adoption of web-bot detection on the internet and its relation to web-bot based studies.

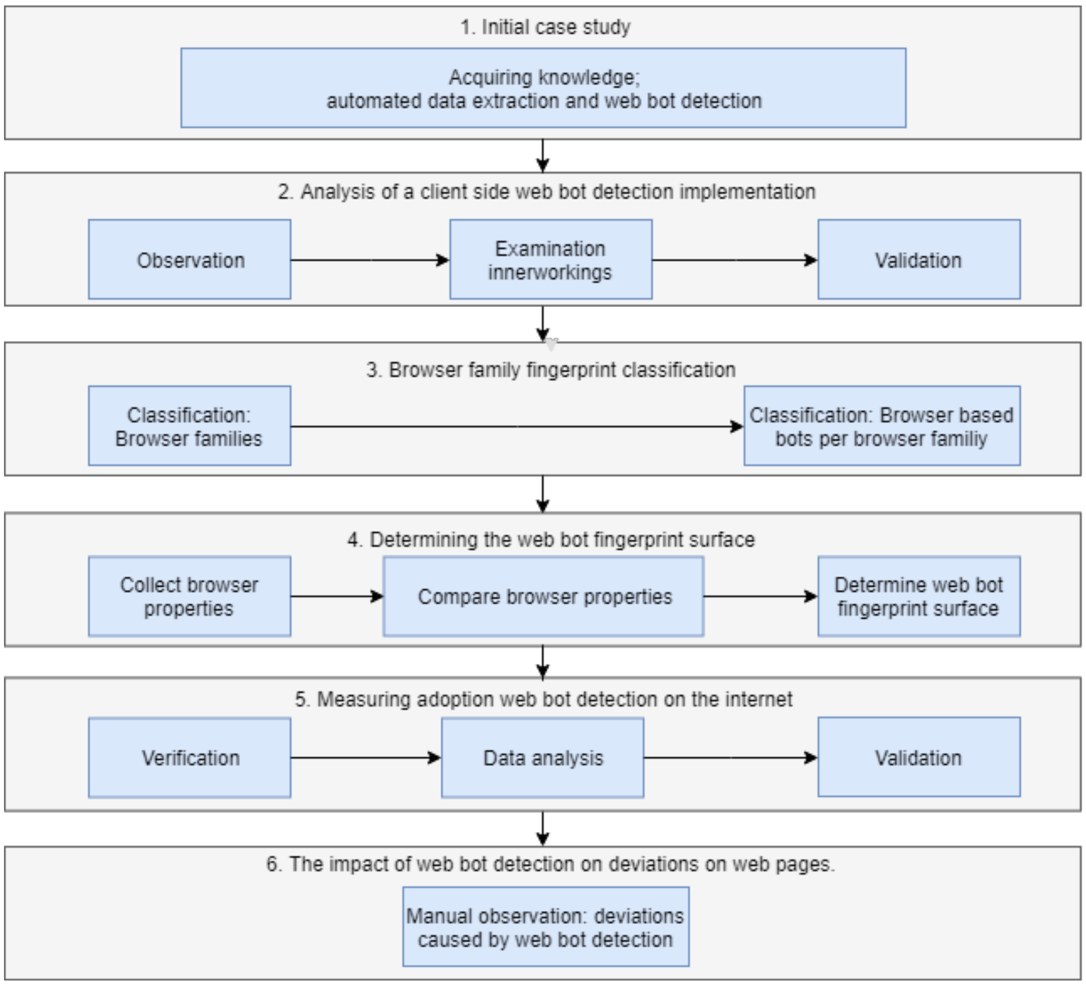

1. Initial case study
Acquiring knowledge; automated data extraction and web bot detection
2. Analysis of a client side web bot detection implementation
Observation
Examination innerworkings
Validation
3. Browser family fingerprint classification
Classification: Browser families
Classification: Browser based bots per browser familiy
4. Determining the web bot fingerprint surface
Collect browser properties
Compare browser properties
Determine web bot fingerprint surface
5. Measuring adoption web bot detection on the internet
Verification
Data analysis
Validation
6. The impact of web bot detection on deviations on web pages.
Manual observation: deviations caused by web bot detection

Figure 4-1: Methodology as a framework

Before we dive into the details of the separated phases of the methodology, we want to stress that we used a browser based approach across all of the phases.
The first reason for this choice is because HTTP based web bots are not relevant in the light of the study proposed (cf. paragraph 2.2). Secondly: browser-based automated data extraction techniques are the techniques most used by scientists (cf. paragraph 2.4).
Reason for this popularity is the capability of using out of the box browser functionalities like navigating on a webpage and accessing the DOM, as well as mimicking the behaviour of a human being. By mimicking browser behaviour, studies tend to extract data that is similar to the data that a human being would see in his or her browser.
Overall; by taking into consideration irrelevant and popular web-bot techniques by using a browser-based approach, we aim for detection of a high coverage of web-bot detection on the internet.

1. Initial case study

We started off our study by conducting an initial case study to gain knowledge about automated data extraction and gain insights in the disadvantages of the employment of web bots in relation to web-bot detection.

The case study is based on a price discrimination project by Krumnow[23] where data has been extracted from specific sites to get a better understanding of price discrimination on the web. The price discrimination project employs a framework that uses PhantomJS, a lightweight headless browser, for extracting content from websites.
During the case study, problems were experienced related to the extraction of data from particular sites using a web bot:

- Particular sites like http://iberia.com will block requests from a particular IP address based on detection of traffic for an unknown amount of time.
- Particular sites like http://iberia.com will block any request from a web cloud based server by default (for instance from Amazon EC2[24]).
- For particular sites like Transavia https://www.transavia.com the exact criteria / algorithm which lead to data deviation is hard to determine.

Additional insights gained during the research showed that the web-bot was not able to extract data from specific sites in particular situations for a specific amount of time.
Also, the location of the web bot, target site and web behaviours like interaction, could lead to data deviations in the form of blocking in respect to a particular page.

Hands-on experience taught us that the problems seem to be very specific in respect to a data extraction approach (cf. appendix A) and even temporary. Data extraction for a particular site could result in blocking pages at one moment in time and could be vanished an hour later.
We concluded that the impediments in relation to automatic data extraction is a real concern and that web-bot detection is hard to prove.
It became obvious that additional research is necessary to prove web-bot detection and to measure the adoption of it on the internet. Sequentially the knowledge can be used to relate web-bot detection to studies based on automated data extraction.

2. Analysis of a client side web-bot detection implementation
By conducting the initial case study, we were convinced of the existence of web-bot detection but we did not have the means to measure web-bot detection on the internet.
To find these necessary means, we decided to analyse an existing web-bot detection implementation to get a better understanding of the used mechanism that we used as foundation to find other web-bot detection implementations. On top of that we were able to prove the existence of web-bot detection based upon its implementation

We chose to analyse a web site with a web-bot that has a strong web-bot detection *smell* based on our observations acquired during the initial case study.
Our goal was to find artefacts of a *client side* detection implementation as it gave us the opportunity to prove web-bot detection this in contrast to a server side detection implementation:

- A client-side approach will be used as we do not possess the code that runs on the server in the context of a web site. Code that is residing on the server can be considered as a black box. A client-side approach is well suitable because a web-bot relates to a web site as a client. In other words; web-bot characteristics are always transferred from client to server as illustrated in Figure 1-1 and Figure 2-1.
- To prove the existence of web-bot detection based on deviations in the contents of web pages based on the *employment* of a web-bot is also hard to do.
  A deviation can be caused by traffic, proxy caching or even A/B testing[25].

3. Browser family fingerprint classification

[23] Benjamin Krumnow , Price Discrimination Project http://www.open.ou.nl/hjo/supervision.htm
[24] https://aws.amazon.com/ec2/
[25] https://www.optimizely.com/optimization-glossary/ab-testing/

The analysis of a web-bot detection implementation taught us that web-bot detection takes place based on specific properties of a web-bot that we will refer to as the *web-bot fingerprint*. In the light of the study we want to detect all possible kinds of web bots.
Aforementioned implies that we have to collect all available web-bot specific properties because a web-bot could get detected based upon those properties.
The only way to gather all specific web-bot properties is to distinguish the properties of a web-bot driven browser from the properties of a human driven browser (which we will refer to as *standalone browser)*.
We have created a browser family classification that we used for the comparison between web-bot and standalone browser properties to prevent that we compare apples with oranges.
For instance, comparing the properties of an Internet Explorer browser to the properties of a Google Chrome web-bot driven browser.

4. Determining the web-bot fingerprint surface
In this phase the browser family fingerprint classification has been used to identify web-bot specific fingerprints that can be used to distinguish a web-bot driven browser from standalone browser which we refer to as the *web-bot fingerprint surface*.
The web-bot fingerprint surface has been used for measuring the adoption of web-bot detection on the internet (phase four).
In line with the choice made for a client side analysis in phase two, we chose to determine the web-bot fingerprint surface based on the properties of a browser.
The first step in phase four concerns the gathering of the browser properties of different standalone and web-bot driven browsers.
Subsequently a delta has been made between the gathered standalone and web-bot driven browser properties by making use of the family fingerprint classification that results in a *web-bot fingerprint surface*. The web-bot fingerprint surface is a union of the browser fingerprint and web-bot fingerprint after validation for usability to measure web-bot detection.
In the context of this study are *browser properties* all properties exposed by a browser instance through the global DOM objects *window, navigator* and *document*.
Browser properties contain *browser characteristics* like for instance the installed plugins, fonts or userAgent but also generic properties of the DOM object itself (e.g. *window.closed* or *navigator.online*).

5. Measuring adoption web-bot detection on the internet
Is this phase the acquired web-bot fingerprint surface acquired during the previous phase are employed to detect web-bot detection on the internet. The web-bot fingerprint surface has been implemented in a scanner embodied in detection patterns that is able to extract script content from a web page and examines the contents on signs of web-bot detection.
The first step in this phase, verification, strengthens the accuracy of the implementation of the scanner by verifying its working on sites containing a strong web-bot detection smell.
The validation step involves the employment of the scanner in the wild to measure the adoption of web-bot detection on the internet. The last step in this phase examines the data gathered by the scanner to measure the coverage of web-bot detection on the internet.

6. The impact of web-bot detection on deviations on web pages
In this phase the effect of web-bot detection on the web sites that are gathered during phase five will be analysed.
We took 20 web sites that we have selected based upon their web-bot detection score and the used web-bot detection patterns.
The main goal is to gain more insights in the different types of deviations that can occur based on web-bot detection. These insights can be used to express the impact of web-bot detection on web-bot based studies.

5 Analysis web-bot detection implementation

This chapter describes the analysis of a web-bot detection implementation on StubHub.com (https://www.stubhub.com).
The site StubHub.com has been chosen as it shows signs of web-bot detection as pointed out by an active discussion on the internet[26].
We found the site simply by searching the internet triggered by the limitations of web bots observed during the initial case study (cf. chapter 4).

The main objective during the analysis was to find artefacts that can be used for the detection of web-bot detection. To fulfil this objective, we first needed to prove the existence of web-bot detection and determine on which implementation level the web-bot detection takes place (cf. paragraph 5.1). After confirmation of the existence of web-bot detection and determination of the implementation level, we investigated the web-bot detection mechanism to determine if the mechanism can be used as foundation for web-bot detection (cf. paragraph 5.2).
This chapter concludes with a validation of the client side detection method (cf. paragraph 5.3).

5.1 Manual observation: browser specific web-bot detection

This paragraph describes the identification of web-bot detection and identifies the layer on which the web-bot detection has been implemented (cf. paragraph 2.1.1) by making use of a manual observation.
We observed web-bot detection by comparing the homepage on StubHub.com between two visits; a visit by a standalone browser and a visit by making use of a web-bot driven browser.

5.1.1 Configuration

For the manual observation we used three different machine configurations for visiting the homepage on StubHub.com:

1. [Manual]: No automated software installed (Windows 10). Google Chrome and Firefox installed.
2. [Selenium + WebDriver]: Selenium with Chrome WebDriver (Linux Ubuntu). Google Chrome and Firefox installed.
3. [Selenium + Plugin]: Selenium IDE Firefox plugin. Firefox and Google Chrome installed

We used Selenium WebDriver[27] and IDE[28] in the role of a web-bot as it is a popular technique that is widely adopted for automated data extraction.

We took into account browser configurations concerning *traffic* and *caching* as it can be used to determine the implementation layer. Determination of the implementation layer is important because it determines if the detection implementation can be used for detecting web-bot detection (cf. chapter 4).

Traffic

A web server may employ web-bot anti patterns if it monitors heavy traffic from a particular client. For instance, multiple HTTP requests within one hour.
A web server is able to trace back the request based upon the IP address of a client.

[26] https://stackoverflow.com/questions/33225947/can-a-website-detect-when-you-are-using-selenium-with-chromedriver
[27] https://www.seleniumhq.org/projects/webdriver/
[28] https://www.seleniumhq.org/projects/ide/

Caching
When a client visits a web page, much information will be cached about the relation between client and the web page. We distinguish two types of cache:

- *Server cache*
  Information stored on the server in relation to the identification of the client. Here we can think of IP addresses in combination with other identification attributes like for instance browser fingerprint.

- *Client cache*
  Information stored on the client, typically in cookies.

Concerning the server side detection mechanisms it is important to distinguish between a 'fresh' web page and a cached webpage. Unfortunately, this distinction is not straight forward as it is hard to determine when a web page is 'fresh'.
A web page can be concerned fresh if the web page has never been visited before from a machine from a particular IP address.
Also, a web page visited from a different IP address that visited the page before can be considered as fresh if the IP variable is not taken into account. In other words, it depends on the target site and its caching mechanisms. When the web page has not been visited by a particular IP address before, we refer to it as 'IP fresh'. In case the client-side cookies are flushed we refer to it as 'Cookie fresh'.

Besides the aforementioned configurations, the following client configuration variables have been taken into account:
- Selenium employment with and without user profile
- Browsing in private and normal mode (Chrome: Incognito, Firefox: Private window).

5.1.2 Analysis
For each configuration the homepage of StubHub.com has been targeted for a maximum of fifteen-page visits within a time window of two days. Manual observation taught us that web-bot detection on StubHub.com is embodied by a web-bot anti-pattern (a Captcha).
We chose to not resolve the Captcha as it could masquerade the effect of the blocking in respect to the other approaches. Besides, resolving could lead to resetting the suspicion of our approaches and configurations that could make it tough to reproduce a blocking behaviour.
The results of the blocking situation are described in Table 5.1.

Machine Configuration	Standalone Google Chrome	Selenium + Chrome WebDriver	Selenium IDE	Max. page visits	Remark
Selenium + WebDriver		✓		2	
	✓			2	Blocking behaviour only occurs *after* the Selenium Chrome WebDriver has been detected.
Selenium + Plugin			✓	2	

Table 5.1: Manual observation web-bot detection StubHub.com

The rows in Table 5.1 illustrating a web page visit per machine configuration. The columns represents the different browser instances used to visit StubHub.com.
Table 5.1 describes that blocking occurs in respect to the configurations; *Selenium + WebDriver* and *Selenium IDE* web-bot detection as indicated with a ✓ (details are described in appendix B).
On both machine web-bot artefacts (automated extraction software) are installed which indicates that StubHub is able to detect them.

Browser specific
A remarkable situation is the blocked standalone browser *Google Chrome* in the first column in relation to the *Selenium + WebDriver* machine configuration.
The reason why this standalone web browser got blocked is most likely because it utilizes the *same* browser that also has been used in case of the *Selenium + Chrome WebDriver* browser instance. That likely the reason why it only got blocked after a visit of the *Selenium + Chrome WebDriver* browser instance.

Irrelevant of browser and network properties
Browser and IP configurations are not described in Table 5.1 as they turned out to be irrelevant in respect to web-bot detection. For instance, it did not matter if we used a 'Cookie fresh' configuration or a private browser mode (cf. appendix B).
The results illustrate that the blocking occurred very configuration specific. For instance, if StubHub.com blocked the page by making use of a Captcha in relation to a specific configuration it did not block the other machine configuration utilizing the same IP.
Aforementioned indicates that blocking does not take place on IP address level.

Based upon the results gained during the manual observations we can deduce that web-bot detection takes place on StubHub.com. Web-bot detection is machine specific in relation to installed web-bot artefacts (Selenium WebDriver / IDE) and browser specific as a standalone browser can also be blocked after detection of a web-bot browser instance.
Network specifics like IP address and caching mechanisms are irrelevant which implies the characteristics for web-bot detection are being determined client side on layer 8 of the OSI model (cf. 2.1.1).

5.2 Web-bot detection based on browser properties

The manual web-bot detection observation described in paragraph 5.1 revealed that web-bot detection takes place on StubHub.com based on a client-side browser based web-bot detection implementation. This paragraph describes an investigation towards the innerworkings of the web-bot detection implementation on StubHub.com to see if the implementation can be used as foundation for detecting web-bot detection. First of all, we investigated the source code for web-bot detection inclusions and found observations for specific browser properties.
Secondly, we analysed the network traffic to find out if the observed browser properties where communicated to the server and triggered web-bot detection.

5.2.1 The observation of browser properties

We investigated the source of the home page on *StubHub.com* for code inclusions related to web-bot detection using the following approach:

1. *Visited https://StubHub.com by making use of Selenium driven web browser.*
2. *Downloaded the page source.*
3. *Extracted inline script content from the page source.*
4. *Extracted URLs linking to external script files and downloading the files.*
   In respect to the external scripts included by the main page we differentiate between first-party scripts and third-party scripts. First-party scripts are scripts originating from the domain of the website (StubHub.com) itself serving the main purpose of the website. The latter originating from different domains or serving a different goal (advertising, social media, user statistics etcetera).
   We have taken into account the first-party and the first level of third-party scripts.

For each script artefacts mentioned in step three and four we have:

5. *De-obfuscated the source by making use of jsbeautifier[29] and code parsing from URI encoded format to its string representation.*
   The title de-obfuscation does not really fit the work of *JsBeautifier. JsBeautifier* is able to reformat the source to make it more readable, unpack scripts packed by specific (Dean Edwards packer[30]) packers and de-obfuscate scripts processed by *javascriptobfuscator[31].*
   If the source is obfuscated by a professional company it will not use *javascriptobfuscator* as it is too familiar to the public. A professional company would prefer to keep the obfuscation algorithm to itself, which decreases the change of effectiveness of *javascriptobfuscator*.
6. *Observed the parsed source for specific keywords.*
   The keywords that we used can be distinguished into the following categories:
   - *Related to cyber security companies*
     During the manual observation (cf. paragraph 5.1) we ran into some URL pointing out to Distil Networks[32].
   - *Web-bot antipatterns*
     Keywords like Captcha, honeypot and the like.
   - *Specific web-bot properties*
     Sources on the internet[33] describe that web bots can be detected by certain keywords. This category represents a collection of mentioned keywords on the web like; *driver, selenium, phantom* and the like.
   - *Global DOM objects*
     The keywords in this category concerning browser properties that uniquely can identify a browser instance. Examination of the occurrence in the source code are important as they can reveal a certain behaviour and a type of browser (web-bot or standalone browser).
     The following Global DOM Objects are being observed:
     - *Navigator*
     - *Browser*
     - *Graphics*

During the source analysis we took into account the possibility that the content of the investigated web page can be tampered with in relation to the Selenium web bot.
For this reason, we compared the responses originating from the different configurations and can confirm that we could not identify web-bot detection related differences across the three configurations.

By conducting the web page source analysis, we have found specific code inclusions towards:

- Web-bot detection
- Browser fingerprinting

Both categories of code conclusions can be found in two different files, respectively: *async.js* and *zwxsutztwbeffxbyzcquv.js* (both can be found in one of the GIT repositories[34] of the project).

Web-bot detection
async.js
In async.js web-bot literals are included in an array obfuscated by a hexadecimal format as illustrated in Code Snippet 5-1.

[29] Jsbeautifier.org
[30] http://dean.edwards.name/packer/
[31] http://javascriptobfuscator.com/
[32] https://www.distilnetworks.com/
[33] For instance; https://stackoverflow.com/questions/33225947/can-a-website-detect-when-you-are-using-selenium-with-chromedriver
[34] https://github.com/GabryVlot/BotDetectionScanner/tree/master/detection/examples/distil

```
var _ac = ["\x72\x75\x6e\x46\x6f\x6e\x74\x73", "\x70\x69\x78\x65\x6c\x44\x65\x70\x74\x68"
, "\x2d\x31\x2c\x32\x2c\x2d\x39\x34\x2c\x2d\x31\x32\x32\x2c",…..]
```

Code Snippet 5-1: Web-bot keyword obfuscated in hexadecimal format (async.js)

After converting the hexadecimal representation to a string, the following web-bot specific literal were found: *$cdc_, phantom, callPhantom, selenium, webdriver* and *driver.*
The keywords are used by their position in the array (for instance *selenium* equals _ac[436]) which made it hard to track down its usage.
For that reason, we have created a small application where we replaced the position reference in the array with the literal in the source as illustrated in Code Snippet 5-2.

```
window[_ac[327]][_ac[635]][_ac[35]](_ac[436])  ===
window[document]["documentElement"]["getAttribute"]("selenium")
```

Code Snippet 5-2: Replacement Web-bot keywords array

The literals originating from the array are being used in three functions called when collecting fingerprint information.
Concerning web-bot detection the function called *Sed()* (which might stands for : Selenium Driver) illustrated in Code Snippet 5-3 is the most relevant function.

```
sed: function() {
    var t;
    t = window["$cdc_asdjflasutopfhvcZLmcfl_"] || document["$cdc_asdjflasutopfhvcZLmcfl_"
] ? "1" : "0";
    var e;
    e = null != window["document"]["documentElement"]["getAttribute"]("webdriver") ? "1"
: "0";
    var c;
    c = void 0 !== navigator["webdriver"] && navigator["webdriver"] ? "1" : "0";
    var n;
    n = void 0 !== window["webdriver"] ? "1" : "0";
    var a;
    a = void 0 !== window["XPathResult"] || void 0 !== document["XPathResult"] ? "1" : "0
";
    var o;
    o = null != window["document"]["documentElement"]["getAttribute"]("driver") ? "1" : "
0";
    var f;
    return f = null != window["document"]["documentElement"]["getAttribute"]("selenium")
? "1" : "0", [t, e, c, n, a, o, f]["join"](",")
}
```

Code Snippet 5-3: Selenium specific web-bot detection function (async.js)

Code Snippet 5-3 describes that the implementation of StubHub.com investigates the properties of the global DOM objects; *window, document* and navigator exposed by DOM api implemented in the browser. The values that are used for validation turn out to be web-bot browser instance specific (cf. chapter 7).

We have executed the function described in Code Snippet 5-3 in context of the different configurations described in paragraph 5.1.
Table 5.2 describes the return value of the function per configuration.

Machine Configuration	Google Chrome	FireFox	Selenium + Chrome Webdriver	Selenium IDE
Manual	0 0 0 0 1 0 0	0 0 0 0 1 0 0		
Selenium + WebDriver	0 0 0 0 1 0 0	0 0 0 0 1 0 0	**1** 0 0 0 1 0 0	
Selenium + Plugin	0 0 0 0 1 0 0			0 0 0 0 1 0 0

Table 5.2: Results execution web-bot fingerprinting function

The rows in Table 5.2 represent the execution of the function per possible browser instance per configuration.
The values in the cell illustrate the return value made out of the characters “0” and “1” that are assigned per web-bot condition within the *sed()* function.
Results indicate that only the Selenium WebDriver scores a “1” for the first condition. The first condition is a check on a specific web-bot property: *$cdc_asdjflasutopfhvcZLmcfl_* which we substantiate in chapter 7.

zwxsutztwbeffxbyzcquv.js
After de-obfuscating of the code in *zwxsutztwbeffxbyzcquv.js*, which involved a high concentration of hexadecimal obfuscated string representation parts, we found several literals referring to web bots as illustrated in Code Snippet 5-4.

```
var driverArray = ["__driver_evaluate", "__webdriver_evaluate", "__selenium_evaluate", "_
_fxdriver_evaluate", "__driver_unwrapped", "__webdriver_unwrapped", "__selenium_unwrapped
", "__fxdriver_unwrapped", "__webdriver_script_function", "__webdriver_script_func", "__w
ebdriver_script_fn"];
var webBotArray = ["_Selenium_IDE_Recorder", "_phantom", "_selenium", "callPhantom", "cal
lSelenium", "__nightmare"];

(arrayInstance["match"](/\$[a-z]dc_/) && window["document"][arrayInstance]["cache_"])

if (!webBotIntegerRepresentation && window["external"] && window["external"].toString() &
& (window["external"].toString()["indexOf"]("Sequentum") != -1))

if ((!webBotIntegerRepresentation) && window["document"]["documentElement"]["getAttribute
"]("selenium"))

    if ((!webBotIntegerRepresentation) && window["document"]["documentElement"]["getAttri
bute"]("webdriver"))

        if ((!webBotIntegerRepresentation) && window["document"]["documentElement"]["getA
ttribute"]("driver"))
```

Code Snippet 5-4: Literals used for detection of automated software (zwxsutztwbeffxbyzcquv.js)

The literals described in Code Snippet 5-4 are being used in a function that is invoked on loading of the web page and after an interval of 400ms.
Also, this function investigates the properties of global DOM objects are being investigated for the presence of a web-bot specific literal.

We executed the aforementioned function in the context of the three configurations described in paragraph 5.1.
Table 5.3 describes the results of the investigation of literals across the used configurations.

System Configuration	Google Chrome	FireFox	Selenium Chrome WebDriver	Selenium IDE
Manual				
Selenium + WebDriver			"$cdc_asdjflasutopfhvcZLmcfl_"	
Selenium + Plugin				"_Selenium_IDE_Recorder"

Table 5.3: General web-bot detection

The results in Table 5.3 describe that the implementation is able to detect a browser instance driven by Selenium WebDriver as well the Selenium IDE plugin in Firefox.
Both values; *$cdc_asdjflasutopfhvcZLmcfl_* and *_Selenium_IDE_Recorder* are web-bot specific properties (cf. chapter 7).

Fingerprinting
In general, we examined that browser fingerprinting and web-bot detection are very close related to each other. For instance, each gesture on the client (by human or bot) browser fingerprinting or web-bot detection takes place based on event handlers as illustrated in Code Snippet 5-5.

```
0 == cf["doadma_en"] && window["addEventListener"] && (window["addEventListener"]("device
orientation", cf["cdoa"], !0), window["addEventListener"]("devicemotion", cf["cdma"], !0)
, cf["doadma_en"] = 1), cf["doa_throttle"] = 0, cf["dma_throttle"] = 0

document["addEventListener"] ? (document["addEventListener"]("touchmove", cf["htm"], !0),
 document["addEventListener"]("touchstart", cf["hts"], !0), document["addEventListener"](
"touchend", cf["hte"], !0), document["addEventListener"]("touchcancel", cf["htc"], !0), d
ocument["addEventListener"]("mousemove", cf["hmm"], !0), document["addEventListener"]("cl
ick", cf["hc"], !0), document["addEventListener"]("mousedown", cf["hmd"], !0), document["
addEventListener"]("mouseup", cf["hmu"], !0), document["addEventListener"]("pointerdown",
 cf["hpd"], !0), document["addEventListener"]("pointerup", cf["hpu"], !0), document["addE
ventListener"]("keydown", cf["hkd"], !0), document["addEventListener"]("keyup", cf["hku"]
, !0), document["addEventListener"]("keypress", cf["hkp"], !0)) : document["attachEvent"]
 && (document["attachEvent"]("touchmove", cf["htm"]), document["attachEvent"]("touchstart
", cf["hts"]), document["attachEvent"]("touchend", cf["hte"]), document["attachEvent"]("t
ouchcancel", cf["htc"]), document["attachEvent"]("onmousemove", cf["hmm"]), document["att
achEvent"]("onclick", cf["hc"]), document["attachEvent"]("onmousedown", cf["hmd"]), docum
ent["attachEvent"]("onmouseup", cf["hmu"]), document["attachEvent"]("onpointerdown", cf["
hpd"]), document["attachEvent"]("onpointerup", cf["hpu"]), document["attachEvent"]("onkey
down", cf["hkd"]), document["attachEvent"]("onkeyup", cf["hku"]), document["attachEvent"]
("onkeypress", cf["hkp"]))
```
Code Snippet 5-5: Device and gesture handlers toward web-bot detection (async.js)

Code Snippet 5-6 describes the invocation of functions declared in an array based on event handlers. The functions invoke indirect the web-bot detection functions described in Code Snippet 5-3 and Code Snippet 5-4 plus functions specific to browser fingerprinting described in Code Snippet 5-6.

Although the construction of a browser fingerprinting is not the main focus of the project described in this document, we observed it in conjunction with web-bot detection.
The results described in Table 5.1 illustrates that also a human 'standalone' driven browser shows signs of blocking after detection of the web bot. This phenomenon has probably been caused by the function described in Code Snippet 5-6.

```
t = this.userAgentKey(t), t = this.languageKey(t), t = this.screenKey(t), t = this.timezo
neKey(t), t = this.indexedDbKey(t), t = this.addBehaviorKey(t), t = this.openDatabaseKey(
t), t = this.cpuClassKey(t), t = this.platformKey(t), t = this.doNotTrackKey(t), t = this
.pluginsKey(t), t = this.canvasKey(t), t = this.webglKey(t), t = this.touchSupportKey(t),
 t = this.videoKey(t), t = this.audioKey(t), t = this.vendorKey(t), t = this.productKey(t
), t = this.productSubKey(t), t = this.browserKey(t), this.keys = t, this.parallel([this.
fontsKey], e)
```
Code Snippet 5-6: Browser fingerprinting functions (zwxsutztwbeffxbyzcquv.js)

5.2.2 Web-bot detection based on client server communication

Paragraph 5.2.1 described that the client-side code implementations on StubHub.com are able to detect a web-bot by observing specific browser properties.
This paragraph describes the data flow running from client to server and server to client to describe how the observation of browser properties results into a deviation on the web page in the form of a Captcha.

Before we dive into the details of the communication between client and server, we first need to take a look at a Distil networks specific code inclusion to get a better understanding of the implemented web-bot detection implementation.

Distil networks specific code inclusion
In the source on StubHub.com we have found a specific *Distil networks* code inclusion that is responsible for displaying a Captcha when suspicious behaviour has been detected and is described in Code Snippet 5-7.
Although the Distil network specific code inclusion in itself is not remarkable (StubHub.com makes use of the services of Distil Networks) it indicates that the server decides if the client should be blocked (based on web-bot detection):

At one point in the code StubHub.com makes a POST request with the header *"X-Distil-Ajax"* to the stubhub.com domain with an interval of 27 seconds.
The response of this post request can contain information to (re) validate the cookie on the server (within the *if* and *else if* statement) or to display a Captcha in case of suspicious client behaviour (*else* statement).
The remainder of this paragraph describes more in depth the contents of a specific request.

```
l("DistilPostResponse");
try {
    var e = r.getResponseHeader("X-UID")
} catch (t) {}
if (document.getElementById("distilIdentificationBlock")) {
    var n = encodeURIComponent(document.location.pathname + document.location.search),
        a = "/distil_identify_cookie.html?httpReferrer=" + n;
    e && (a = a + "&uid=" + e), document.location.replace(a)
} else if (document.getElementById("distil_ident_block")) {
    var i = "d_ref=" + document.location.pathname.replace(/&/, "%26");
    i += "&qs=" + document.location.search + document.location.hash, e && (i = "uid=" + e
 + "&" + i), document.location.replace("/distil_identify_cookie.html?" + i)
} else(document.getElementById("distil_ident_block_POST") || document.getElementById("dis
tilIdentificationBlockPOST")) && window.location.reload()
```

Code Snippet 5-7: Distil specific code observing post response for revalidation or blocking of the client (zwxsutztwbeffxbyzcquv.js)

Network analysis
We used a proxy to monitor all traffic running from client to server. Although we have used a proxy that is able to monitor different protocols it turned out that the most interesting traffic concerned HTTP traffic. For that reason, we will only discuss HTTP requests and responses in this paragraph.
Network traffic has been analysed by visiting StubHub with the configurations *Manual* and *Selenium + WebDriver* for describing the difference between a web-bot driven browser instance and a standalone browser instance.
During the observation, StubHub.com has been visited using the Chrome browser to a maximum of three requests.
For the sake of clarity, the HTTP request/responses are narrowed down to the requests related to the source files mentioned in paragraph 5.2.1.
Concerning the configuration *Selenium + WebDriver*; the observation took place after resolving the Captcha from the same machine as used for gathering of the results displayed in Table 5.1.

Diagram 5-1 illustrates the flow of HTTP requests of the first visit regarding both configurations in a sequence diagram between web server (StubHub.com) and browser.
Table 5.4 gives a description of the noteworthy requests.

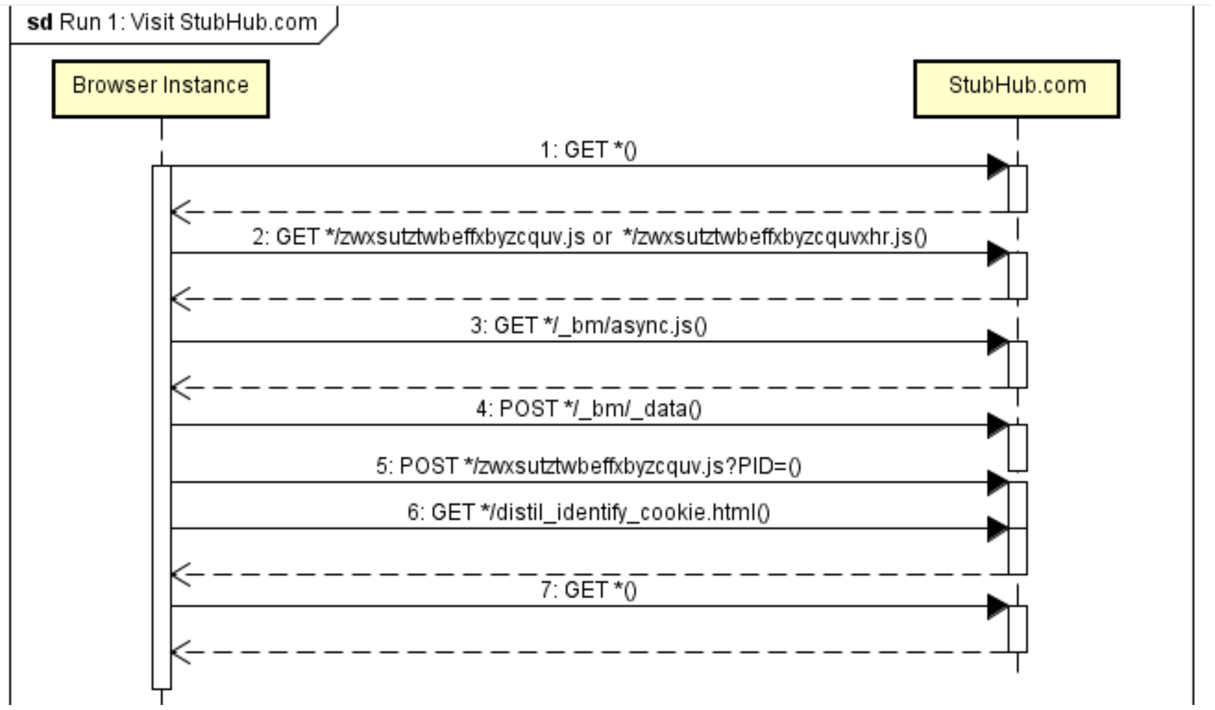

sd Run 1: Visit StubHub.com
Browser Instance
StubHub.com
1: GET *()
2: GET */zwxsutztwbeffxbyzcquv.js or */zwxsutztwbeffxbyzcquvxhr.js()
3: GET */_bm/async.js()
4: POST */_bm/_data()
5: POST */zwxsutztwbeffxbyzcquv.js?PID=()
6: GET */distil_identify_cookie.html()
7: GET *()

Diagram 5-1: HTTP requests first visit StubHub.com

Request	Description
1	Requests the main content of StubHub.com. The main content of the *manual* configuration points to an external file ‘zwxsutztwbeffxbyzcquv.js’ in contrast to the configuration *Selenium + WebDriver* that points to an external file with the name ‘zwxsutztwbeffxbyzcquv**xhr**.js’. In both cases the main content contains a *div* element containing the id *distilIdentificationBlock* as illustrated in Code Snippet 5-7.
2	Zwxsutztwbeffxbyzcquv**xhr**.js does not contain the function that is able to detect web bots this in contrast to zwxsutztwbeffxbyzcquv.js The XHR version does apply to configuration *Selenium + WebDriver* which lacks the function for detecting automated software and feels for that reason a bit contradictory. We could not find an explanation for this behaviour in the client-side sources but we suspect that earlier during manual observation (cf. paragraph 5.2.1) made the web-bot detection function superfluous.
4	Body of the request contains fingerprint information including the web-bot information from *async.js* as described in paragraph 5.2.1. The gathered browser fingerprint information together with the web-bot indication will be sent (obfuscated) to the server.
5	Body of the request contains fingerprint information including the web-bot information from *zwxsutztwbeffxbyzcquv.js.* This is without the web-bot information. Also, the body contains a literal (“proof”) from a Distil specific piece of code.
6	Cookie identification
7	Body of the response contains the full main page content.

Table 5.4: Description HTTP request first visit StubHub.com

To summarize the findings based on the HTTP request of the first run:
The server host is aware of a lot of details of the client:
- Browser fingerprint
- Web-bot fingerprint
- Cookie settings

The server has all the means to place a web-bot specific deviation on the web page.

Additional general remarks:
- By observing the HTTP traffic, it is likely that the web-bot detection code in *zwxsutztwbeffxbyzcquv.js* is not being executed on loading of the main page.
- The last GET request towards the root URL does not contain signs of blocking while the client could have been remarked as suspicious. One reason for this behaviour might be time between the POST requests and the last GET request in the table. This suspicion is being strengthen by the fact that after a click on a link on the page the page is being blocked by a Captcha.

The second run involves clear differences between the different requests from both configurations. Table 5.5 and Table 5.6 illustrating the requests and responses of the second page visit with respectively *Manual* and *Selenium + WebDriver*.

URL	Type	Request remarks	Response remarks
*	GET	Cookie included	Response contains source of the main page.
*/zwxsutztwbeffxbyzcquv.js	GET	Cookie included	
*/zwxsutztwbeffxbyzcquv.js?PID=	POST	Cookie included. Body of the request contains fingerprint information including the web-bot information from *zwxsutztwbeffxbyzcquv.js* as described in paragraph 5.2.1.	

Table 5.5: HTTP requests and responses run 2 Manual

* https://www.stubhub.com/

URL	Type	Request remarks	Response remarks
*	GET	Cookie included	Response contains a body with the **Captcha** page containing a link to the external script file *distil_r_captcha_util.js*
*/distil_r_captcha_util.js	GET	Cookie included	Response body contains Captcha specific code containing a XmlHttpRequest (XHR).
*/zwxsutztwbeffxbyzcquv.js	GET	Cookie included	
*/zwxsutztwbeffxbyzcquv.js?PID=	GET	Cookie included. Body of the request contains fingerprint information including the web-bot information from *zwxsutztwbeffxbyzcquv.js*	

		as described in paragraph 5.2.1. Also, the body contains a literal ("proof") from a Distil specific piece of code.	

Table 5.6: HTTP requests and responses run 2 Selenium + WebDriver

* https://www.stubhub.com/

The most remarkable about the second run is that the Selenium driven browser instance is being blocked as cause of the response content of the first GET request to the root URL. Despite the usage of a cookie the request, the response still contains a body with the Captcha after first clearing of the browser cookies.
Aforementioned indicates that the browser instance has been marked as suspicious at the server.
The third run does not have any significant differences compared to the second run regarding web-bot detection.

5.3 Validation of the client side web-bot detection

As described in paragraph 2.1.1: web-bot detection can be implemented on different implementation levels and therefore different web-bot detection mechanisms can be in place. To determine the usage of the web-bot detection mechanism implemented on StubHub.com as foundation for detecting web-bot detection, we must find out if it is relevant.
In other words; is the client side web-bot detection implementation as described in paragraph 5.2 the cause of the deviation experienced on StubHub.com?
To prove its relevance and answer the aforementioned question positive, we used a counter measure to exclude web-bot detection on other implementation levels.

The countermeasure is based on the configuration: *Selenium + WebDriver* which concerns the detection of the Selenium web driver. The reason for the focus on configuration *Selenium + WebDriver* is based upon the fact that Selenium web driver is a very popular and widely used framework regarding automatic data extraction, which stands in contrast to Selenium IDE that is in fact entered the 'end of life' software cycle[35].
The implementation of the countermeasure is based upon a hint mentioned on the Web-bot Selenium Detection post on *StubHub.com*[36].

In Code Snippet 5-3 and Code Snippet 5-4 we described that the Selenium web driver browser instance can be detected by the literal: *$cdc_asdjflasutopfhvcZLmcfl_*.
After research guided by the StubHub.com[37] post it turned out that this literal originates from the ChromeDriver[38].
The Chrome driver is a piece of software that makes it possible to automatically drive a chromium / Google Chrome browser.
ChromeDriver is part of the Chromium project and lives in the Chromium repository. Chromium is the open source project which Google Chrome is based on.
Because of the fact that it is an opensource project we did have access to the code of the Chrome Driver. By checking out the Chromium project on our local machine (by using the instructions described on the Chromium project site[39]) we starting searching for the specific *$cdc* literal.

[35] https://seleniumhq.wordpress.com/2017/08/09/firefox-55-and-selenium-ide/
[36] https://stackoverflow.com/questions/33225947/can-a-website-detect-when-you-are-using-selenium-with-chromedriver
[37] https://stackoverflow.com/questions/33225947/can-a-website-detect-when-you-are-using-selenium-with-chromedriver
[38]
[39] https://chromium.googlesource.com/chromium/src/+/master/docs/linux_build_instructions.md

Eventually we found the literal in the function as illustrated in *Code Snippet 5-8* residing in the file *test/chromedriver/js/ call_function.js* within the source of the Chromium project.

```
function getPageCache(opt_doc, opt_w3c) {
    var doc = opt_doc || document;
    var w3c = opt_w3c || false;
      var key = '$cdc_asdjflasutopfhvcZLmcfl_';
    if (w3c) {
        if (!(key in doc)) doc[key] = new CacheWithUUID();
        return doc[key];
    } else {
        if (!(key in doc)) doc[key] = new Cache();
        return doc[key];
    }
}
```

Code Snippet 5-8: $cdc detection literal in ChromeDriver source

We changed the value of the key with a random string with a length of 30 characters that is being generated only once when the function will be invoked for the first time.
The usage of a random key ensures that the key cannot (anymore) be used for detecting the automatically driven chrome browser instance.
Only its length is detectable which is not a very strong property to check for.
The reason why we only generated it once is because we wanted to avoid that the functionality of the Chrome driver brakes when the value of the key will be referenced within the browser.

After making the modifications in the source file, we compiled the ChromeDriver with the aforementioned changes.
Sequentially we used this custom ChromeDriver in configuration *Selenium + WebDriver* and tested the web-bot detection methods described in Code Snippet 5-3 and Code Snippet 5-4 in the browser console.
It turned out that bot detection functions in respectively *async.js* and *zwxsutztwbeffxbyzcquv.js* were not able to detect the *$cdc* literal.
We also visited StubHub.com 30 times without being blocked which validated that StubHub.com indeed makes use of client side web-bot detection based on specific web-bot properties.

5.4 Summary

This chapter described that web-bot detection takes place client side on StubHub.com. By observing the behaviour of the web site, we concluded that the web-bot detection is machine and browser specific and irrelevant of network properties or caching.
By analysing the innerworkings we found out that browser fingerprinting takes place together with web-bot detection. Both are based on browser properties and is (obfuscated) being sent to the server host triggered by user gestures and time intervals
The code responsible for the web-bot detection itself is also obfuscated and focuses on browser properties that are web-bot specific (web-bot fingerprint).

We can conclude that the observed web-bot detection implementation can be used as foundation for detecting web-bot detection because it takes place client side and uses specific browser properties which makes web-bot detection observable.

6 Browser family fingerprint classification

Chapter 5 described a specific web-bot detection implementation based on browser properties that can be used as foundation for the detection of web-bot detection. The web-bot implementation described, uses specific web-bot browser properties for the identification of a web bot.

In the light of the study proposed we want to detect as many web-bot detection implementations as possible. Aforementioned implies that we needed to identify all possible web-bot specific browser properties that can be used to detect web-bot detection implementations on the internet.

We achieved this goal by answering the following question: *What are the specific web-bot browser properties of a web-bot driven browser instance?*

To answer this question, we need to find browser bot identification properties that can be used to distinguish a web-bot from a standalone browser. By comparing the properties of a standalone browser with a web-bot driven browser we can find the browser bot identification properties.

Different browser types like for instance, Mozilla Firefox and Google Chrome, have different browser properties. Aforementioned also applies to web-bot driven browser instances that are based upon a specific browser type.
To avoid comparing oranges to apples, we first classified browser families, as described in paragraph 6.1.
Paragraph 6.2 introduces web-bot frameworks and describes their relation to browser families.
This chapter concludes with paragraph 6.3 that gives an overview of the browser coverage used in this study in relation to the browser family classification.

6.1 Browser families

There are a lot of different browser types that are being used to extract data from the content of a webpage.
The intention of the browser family classification described in this chapter is not to give a complete overview of all used browser and browser engines in respect to data extraction, but instead it focuses on the most used browsers[40] that also can be used by as a web-bot driven browser instance.

The browser family classification is based upon commonalities grounded in the innerworkings of a browser by their layout engine[41] (or web browser engine) and JavaScript engine[42]. The layout engine is responsible for rendering the web pages where the JavaScript engine is responsible for the execution of JavaScript code within a webpage. In the context of the study we refer to combination of used layout and JavaScript engine as the *browser family fingerprint* that uniquely can identify browser family.

Table 6.1 and Table 6.2 describing respectively; layout and JavaScript engines in relation to popular web browsers.

Layout engine	Browser	Browser version	Development	Remarks
WebKit	Google Chrome	<= 26 and mobile version for the IOS system.	Apple, Adobe Systems, Google, KDE and others	
	Safari		Apple	

[40] https://www.w3counter.com/globalstats.php
[41] https://en.wikipedia.org/wiki/Web_browser_engine
[42] https://en.wikipedia.org/wiki/JavaScript_engine

	Firefox for iOS		Mozilla foundation	
Blink	Google Chrome	>= 27	Google	Fork of Webkit's WebCore Component
	Chromium		The Chromium Project	Open source project started by Google. Chromium shares the majority of code and features with Google Chrome.
	Opera	>=15	Opera Software	
Gecko	Firefox		Mozilla foundation	
Trident	Internet Explorer		Microsoft	
EdgeHTML	Edge		Microsoft	Fork of Trident
Presto	Opera	<=14	Opera Software	

Table 6.1: Layout engines in relation to their usage in web browsers.*

** Source: Wikipedia (For a more extensive list take a look at the list of web browsers in combination with layout engine usage on Wikipedia[43]).*

JavaScript Engine	Browser	Development	Remarks
JavaScriptCore	Safari	Apple	Component of WebKit
V8	Google Chrome	The Chromium Project	
	Chromium		
	Opera		
SpiderMonkey	Firefox	Mozilla foundation	
JScript	Internet Explorer	Microsoft	
Chakra	Edge	Microsoft	Chakra is a fork of JScript

Table 6.2: JavaScript engines in relation to their usage in web browsers.*

** Source: Wikipedia[44]*

The layout and JavaScript engines described in Table 6.1 and Table 6.2 illustrate that browser categories can be distinguished based upon their innerworkings. For example, if a web site is able to identify the *Trident* layout engine, then the chance of dealing with the Internet Explorer browser is high. In relation to web-bot detection those characteristics are important to the identification of particular web bots as described in paragraph 6.2.

6.2 Browser based web bots

This paragraph describes frameworks that can be employed as web-bot driven browser instances to extract data from a webpage.
The purpose of this paragraph is to give an introduction of popular web-bot frameworks and (if applicable) to describe the differences in respect to a standalone browser.

[43] https://en.wikipedia.org/wiki/List_of_web_browsers
[44] https://en.wikipedia.org/wiki/JavaScript_engine

The content of this paragraph is limited to the most used frameworks in studies, which are: *PhantomJS*[45]*, CasperJS*[46]*, SlimerJS*[47]*, Selenium WebDriver*[48]*, Headless Chrome*[49] and *NightmareJS*[50].

PhantomJS
PhantomJS is a headless browser (cf. appendix A) used for automating web page interaction and is based on *WebKit*. PhantomJS provides a JavaScript API enabling automated navigation, screenshots, user behaviour and assertions.
PhantomJS is a scriptable browser which means that the browser can be accessed programmatically. PhantomJS is not a full-fledged browser with hundreds of engineers working on it. PhantomJS has its limitations as describes in the work of Englehardt et al. [EN16].

CasperJS
CapserJS is a suite of libraries on top of PhantomJS that extend its capability.

SlimerJS
SlimeJS is similar to PhantomJS except that it runs on top of *Gecko* instead of *WebKit.* SlimerJS is not truly headless until version 56 of Firefox, because *Gecko* cannot render web content without a graphical window.

Selenium WebDriver
Selenium WebDriver is a prominent component of Selenium[51] which is a portable software-testing framework for web applications. Selenium WebDriver drives a browser by accepting command and sending them to a browser. This technique is implemented through a browser-specific browser driver which sends commands to a browser and retrieves results.
The definitions of a WebDriver as given by the World Wide Web Consortium (W3C)[52]:

"WebDriver is a remote control interface that enables introspection and control of user agents. It provides a platform- and language-neutral wire protocol as a way for out-of-process programs to remotely instruct the behavior of web browsers."

There are different WebDrivers available for different browser like *Chrome (Google Chrome / Chromium)*, *Firefox, InternetExplorer, Edge, PhantomJS, Safari* and the like.

By using a WebDriver implementation, Selenium WebDriver is able to drive a full-fledged browser.

Headless chrome
Chrome / Chromium can programmatically be run and driven in a headless environment by making use of the *chrome-remote-interface*[53] (or the Puppeteer API[54]). The *chrome-remote-interface* is a Node library to control headless (or full) Google Chrome or Chromium. By making use of the *chrome-remote-interface* the situation is similar to PhantomJS although it uses a full-fledged browser.

NightmareJS
NightmareJS is a high-level browser automation library running on *Electron*[55]. Electron is a framework for crating native application based on web technologies.

[45] http://phantomjs.org/
[46] http://casperjs.org/
[47] https://slimerjs.org/
[48] http://www.seleniumhq.org/projects/webdriver/
[49] https://chromium.googlesource.com/chromium/src/+/lkgr/headless/README.md
[50] http://www.nightmarejs.org/
[51] https://en.wikipedia.org/wiki/Selenium_(software)
[52] https://www.w3.org/TR/webdriver/
[53] https://github.com/cyrus-and/chrome-remote-interface
[54] https://developers.google.com/web/tools/puppeteer/
[55] https://electronjs.org/

NightmareJS exposes simple methods that mimic user actions by making use of an API that drives the Chromium web browser on the front-end. By making use of the provided API, scripts are able to drive Chromium programmatically.

6.3 Browser family classification

This paragraph describes the browser family classification based upon the browser family fingerprint that is being used to identify specific web-bot browser properties as will be described in chapter 7.

In respect to the restricted time of the study we are not able to take all of the described browsers in Table 6.1 and Table 6.2 into consideration.
Due to the high coverage in use and the available support for web-bot driven instances we chose to pick the following browsers: *Google Chrome, Chromium, Firefox, Internet Explorer and Edge* where the version number of Google Chrome and Chromium are higher than 26.

Table 6.3 describes the browsers family classification based upon their browser family fingerprint of the browsers that are covered in this study.

Browser family	Browser family fingerprint		Standalone browser	Browser based web bot
	Layout Engine	JavaScript engine		
Blink + V8	Blink	V8	Google Chrome / Chromium	Selenium + WebDriver, Headless Chrome, PhantomJS* (CasperJS, SlimerJS), NightmareJS
Gecko + Spidermonkey	Gecko	SpiderMonkey	Firefox	Selenium + WebDriver
Trident + JScript	Trident	JScript	Internet Explorer	Selenium + WebDriver
EdgeHTML + Chakra	EdgeHTML	Chakra	Edge	Selenium + WebDriver

Table 6.3: Browser study coverage and their layout and JavaScript engine commonalities

* PhantomJS is based on WebKit which shares the WebCore component with Blink based layout engines.

Each row in table 6.3 represents a browser family based upon the browser family fingerprint deduced from the combination of layout and JavaScript engine described in the second column.
Respectively, the column 'Standalone browser' and the column 'browser based web bot' contain browsers that can be driven manually and can be driven by a web-bot belonging to the applicable browser family.

Each browser family will be used to compare the browser properties of a standalone and web-bot driven browser instance which each other to gather specific web-bot browser properties (cf. chapter 7).

7 Determining the web-bot fingerprint surface

This chapter describes how we determined *the web-bot fingerprinting surface* by making use of the browser family classification described in paragraph 6.3.
The web-bot fingerprinting surface are specific browser properties that can be used to detect a web bot.

Paragraph 7.1 describes the relevant browser properties that will be used to identify web-bot specific web-bot properties.
Paragraph 7.2 describes the approach of how we obtained the web-bot specific browser properties described in paragraph 7.3.
Paragraph 7.4 analysis the obtained web-bot specific browser properties and classifies only the relevant properties which will lead to the web-bot fingerprinting surface.

7.1 Relevant browser properties

In this study we solely focus on client side web-bot detection based on the properties exposed by a browser of the following reasons:

- Web-bot properties can be used to prove the existence of web-bot detection because they are detectable client side which makes them more reliable in contrast to network traffic analysis or server log analysis.
- It is more likely that server hosts employing web-bot detection mechanisms based on web-bot properties client side rather than server-side.
- Server hosts can only determine if the client is a web-bot based on the information that comes from the client. The client is also a black box for the server which means that relevant information regarding web-bot detection must come from the client. Aforementioned implies that web-bot detection takes place on the client likely based on the properties of a web bot.
- State of the art automated software is very well capable of mimicking the behaviour of a human being which makes the detection of a web-bot purely based on network activity nearly impossible.
- Code that runs on the server is a black box and can therefore not be used to observe web-bot detection.

Aforementioned implies that communication characteristics (communication with the HTTP protocol between browser and webserver) will not be taken into consideration as they cannot be measured by server hosts on the client.

The intention is to identify browser properties that are likely being used for web-bot detection by web sites. Chapter 5 described a specific web-bot detection implementation based on specific web-bot browser properties. Unfortunately, we do not have knowledge about specific web-bot properties exposed by a web browser so we need to observe a wide area of browser properties.
Properties that fall under the flowing classification will be examined on web-bot detection relevance by the approach described in paragraph 7.2.

- **Browser**
  Properties in this category reveal browser characteristics like for instance installed plugins, user-agent, language, cookies settings, used fonts, details about the layout and JavaScript engine and the like.

- **Screen**
  Screen properties describe characteristics about the visualization of a rendered web page like screen width, height and colour depth.

Remark that the screen width and height (its resolution) can easily be configured by a web bot. For that reason, we do not take solely the value of those characteristics into consideration but compare those values in relation to other properties like max allowed width and height.

- **Context**
  Context properties tell something about the context of the running browser instances. User profiles, user data and attributes related to the control of the browser (web-bot versus human driven) belong to this category.

7.2 Approach: obtaining deviating browser properties

This paragraph describes the approach for obtaining web-bot specific browser properties by making use of the browser family classification described in paragraph 6.3.

Web-bot browser properties are obtained by comparing the properties of a web-bot driven browser instance with a standalone browser of the same browser family.
Table 7.1 describes the *compare configurations* based on browser families that have been used to compare the browser properties of a web-bot driven and standalone browser instance.

Browser family	Standalone browser	PhantomJS	Selenium WebDriver (Google) Chrome)	Selenium WebDriver (Chromium)	Selenium WebDriver (Firefox)	Selenium WebDriver (Internet Explorer)	Selenium WebDriver (Edge)	NightmareJS	Headless Chrome
Blink + V8	Google Chrome	✓	✓	✓				✓	✓
	Chromium								
Gecko + SpiderMonkey	Firefox				✓				
Trident + JScript	Internet explorer					✓			
EdgeHTML + Chakra	Edge						✓		

Table 7.1: Compare configurations between web-bot and standalone browser instances

The comparisons described in Table 7.1 are based on the inner workings of the browser (cf. chapter 5). This means for instance that all browsers based upon the Blink Layout engine and V8 JavaScript engine are compared against each other (indicated by the ✓ symbol).
By comparing web-bot and standalone browser instances, for instance, PhantomJS and a user driven Chrome browser, web-bot specific fingerprinting properties can be revealed.
The same applies to comparing the different web-bot driven browsers by Selenium. Similarities between the specific web driven browsers fingerprint attributes will filter the false negative attributes and strengthen the true positives.

For each comparison one particular standalone browser per browser family has been taken as baseline. We have to make a remark that we have chosen Google Chrome over Chromium in respect of the Blink / V8 browser family because of the following two reasons:

1 Google Chrome contains a richer set of browser attributes as Chromium serves a base for Google Chrome.
2 The differences in browser attributes between the two browsers are negligible as illustrated in Table 7.2. Note: In the case of Chromium the Flash player could not be installed.

	Google Chrome	Chromium
User Agent	Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/64.0.3282.140 Safari/537.36	Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) ***Ubuntu Chromium/64.0.3282.167 Chrome/64.0.3282.167 Safari/537.36***
Plugins	• Chrome PDF Plugin::Portable Document Format::application/x-google-chrome-pdf~pdf • Chrome PDF Viewer::::application/pdf~pdf • Native Client::::application/x-nacl~ application/x-pnacl~ • Shockwave Flash::Shockwave Flash 29.0 r0::application/x-shockwave-flash~swf • application/futuresplash~spl • Widevine Content Decryption Module::Enables Widevine licenses for playback of HTML audio/video content. (version: 1.4.9.1070)::application/x-ppapi-widevine-cdm~	• **Chromium** PDF Plugin::Portable Document Format::application/x-google-chrome-pdf~pdf • **Chromium** PDF Viewer::::application/pdf~pdf • Widevine Content Decryption Module::Enables Widevine licenses for playback of HTML audio/video content. (version: **undefined**)::application/x-ppapi-widevine-cdm~

Table 7.2: Differences in fingerprinting attributes: Google Chrome compared to Chromium

Browser properties
The browser properties itself are being collected by making use of active fingerprinting as described in the work of Torres et al. [TJM14].
Active fingerprinting involves gathering the browser properties of a specific browser instance by making use of client-side scripting.
Typically, JavaScript has being used to obtain browser properties related to the employment of browsers by targeting the following global DOM objects exposed by a browser:

- **window**[56]
  The window object represents an open window in a browser. A window object can contain other nested objects. The window object contains attributes that fall into the *screen* classification mentioned in paragraph 7.1.
- **navigator**[57]
  The navigator object contains information about the browser.
- **document**[58]
  The document object represents a HTML document which can contain other nested documents. A webpage is at least represented by one document object. By accessing the document object attributes regarding a rendered page can be obtained.

We used the fingerprinting library *Fingerprintjs2*[59] for obtaining browser fingerprinting properties extended with the following custom browser properties:

- **Names of the root properties of the *window* and *document* object.**

[56] https://www.w3schools.com/jsref/obj_window.asp
[57] https://www.w3schools.com/jsref/obj_navigator.asp
[58] https://www.w3schools.com/jsref/dom_obj_document.asp
[59] https://github.com/Valve/fingerprintjs2

Code Snippet 5-3 and Code Snippet 5-4 illustrated that additional properties on the global *window* and *document* browser objects can reveal the employment of a web bot.

- **The attributes of the *navigator* object.**
  The global *navigator* browser object can contain information about the driver of the browser (cf. Code Snippet 5-3). Not only the existence of a particular property is important but also its content.
- **Lack of *bind* JavaScript engine feature.**
  Particular Web bots make use of older browser JavaScript engines. By checking the *bind* JavaScript feature, older web bots (like for instance PhantomJS 1.x and older[60]) can be detected.
- **StackTraces**
  StackTraces thrown by the JavaScript engine can show signs of the employment of a web bot.
- **Missing image**
  In case of headless browsers (often driven in the employment of a web bot) an image will be loaded with a width and height equal to zero[61].
- **Web security (ability to execute cross side scripting)**
  Web bots are notorious for execution of cross side scripting which requires that the web security browser attribute has been set to false.
- **Suppression of popups[62]**
  Especially headless browsers have a build in mechanism to automatically suppress popups (JavaScript alerts). By detecting this behaviour, a web-bot could be distinguished form a standalone browser.

Configuration

We used a web page[63] for the extraction of browser properties inspired by the work of Eckersley [ECK10] to extract and persist the browser properties of web bots driven and standalone browser instances. Eckersley used a web site to investigate the unique identification of a browser based upon the unique fingerprint of a web browser.
The determining of the web-bot fingerprinting surface is strongly related because it also uses a web page that targets the properties of a web browser.
A dedicated web site to extract and persist browser properties offered us two advantages:

1. We were able to mimic the behaviour of a server host by applying active fingerprinting without meddling the server host.
2. There was no risk involved that the properties of the browser were tampered with.

We visited the test website by making use of 11 different browser configurations as described in Table 7.3.

Configuration	Version	OS
PhantomJS	2.1.0	Ubuntu 16.04
Google Chrome	64.0.3282.140	Ubuntu 16.04
Chromium	64.0.3282.140	Ubuntu 16.04
Selenium + Google Chrome	4.0.0	Ubuntu 16.04
Selenium + Chromium	4.0.0	Ubuntu 16.04
Selenium + Firefox	4.0.0	Ubuntu 16.04
Mozilla Firefox	58.0.2	Ubuntu 16.04
Microsoft Edge	41.16.299.15.0	Windows 10

[60] https://blog.shapesecurity.com/2015/01/22/detecting-phantomjs-based-visitors/
[61] https://antoinevastel.com/bot%20detection/2017/08/05/detect-chrome-headless.html
[62] https://www.slideshare.net/SergeyShekyan/shekyan-zhang-owasp
[63] https://github.com/GabryVlot/BrowserBasedBotFP

Internet Explorer	11.0.50	Windows 10
Nightmare	2.10	Ubuntu 16.04
Automated chrome	Puppeteer 1.1.0 Google Chrome and Chromium	Ubuntu 16.04

Table 7.3: Used browser configurations for visiting browser properties extraction website.

Obtaining browser bot identification properties
After loading of the web page, the properties of the browser are being captured by making use of active fingerprints and are being send back to the server to persist the properties.
An overview of all the involved activities is illustrated by the sequence diagram in Diagram 7-1.

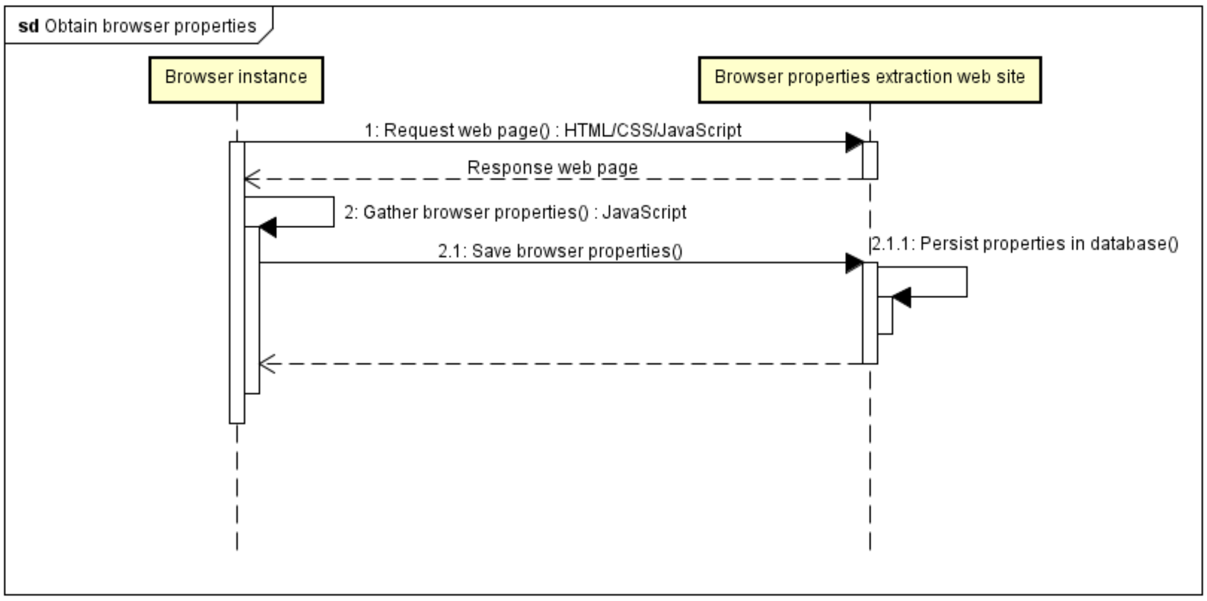

sd Obtain browser properties
Browser instance
Browser properties extraction web site
1: Request web page() : HTML/CSS/JavaScript
Response web page
2: Gather browser properties() : JavaScript
2.1: Save browser properties()
2.1.1: Persist properties in database()

Diagram 7-1: Persisting of browser properties

After persisting all of the relevant browser properties mentioned in section *browser properties* we created a delta by making use of the compare configurations described in Table 7.1. For creating the delta, we used a script file that loops through the properties of each browser instance which resulted in web-bot specific browser properties which are described in paragraph 7.3.

7.2.1 Design improvements

In respect to the *browser properties extraction* web page: we strived for the design of a modular and re-usable application by abstracting the obtaining of the browser properties away from the application (the web page).
The obtaining of the browser properties resides in a separated module containing a dedicated function for each type of browser property (cf. section browser properties).
Unfortunately, there exists a tightly coupled construction between the main application and the gathering of the fingerprints, embodied by the invoking of the methods in the browser property module by the main application.
This leaves room for improvement which due to time limitations could not be conducted. Concerning the extendibility of (possible) future relevant browser properties the application will benefit from a plugin architecture.
In a plugin architecture, a specific *browser property extractor* can serve as a plugin that conforms a specified *interface* which can be dynamically be loaded by the main application.
The aforementioned architecture makes the application easy extendable, maintainable and re-usable by forcing a loosely coupled implementation between property extraction and the main application.

7.2.2 Limitation

The approach for obtaining web-bot specific browser properties described in this paragraph is based upon specific data extraction software versions (cf. section *configuration*).
Using specific software versions is a limitation as we might not be able to detect web-bot browser properties of older or newer software versions.
Consider a hypothetical situation where Selenium version 1 exposes a Selenium specific property A and selenium version 3 exposes a Selenium specific property C.
If we compare a standalone browser with a web-bot driven browser instance based on a Selenium version 2 we might not be able to detect the web-bot specific properties A and C.
Although we did not take into account different versions of data extraction software it is good to realize that there might be more web-bot specific browser properties.
Using a wider range of data extraction software can be the first valuable step toward future work.

7.3 Deviating browser properties

This paragraph describes the results of the comparison between the browser properties of standalone and web based driven browsers by making use of the approach described in paragraph 7.2.

We described the results of each comparison per browser family in a separated table as described in Table 7.4.

Browser family	Table
Blink + V8	Table 7.5
Gecko + Spidermonkey	Table 7.7
Trident + Jscript	Table 7.6
EdgeHTML + Chakra	Table 7.8

Table 7.4: Overview fingerprinting results per browser family category.

The first column of each of the tables described in Table 7.4 lists the deviating browser properties after comparison of the browser properties of web-bot driven browsers (represented by column three and if applicable the successive columns) and the standalone browser described in the second column.
We excluded properties that cannot be detected client side (for instance HTTP headers).
A ✓ indicates that the attribute value differs from the standalone browser property and a *x* indicates that the property could not be determined.
Details about the specific differences are described in appendix C.

Standalone browser: Google Chrome	PhantomJS	Selenium + Google Chrome		Selenium + Chromium		NightmareJS	Automated Google Chrome		Automated Chromium	
		Headless	**Headful**	**Headless**	**Headful**	**Headless / Headful**	**Headless**	**Headful**	**Headless**	**Headful**
User Agent*	✓	✓				✓	✓			
Color depth	✓									
Hardware concurrency	*x*									
Resolution	✓	✓		✓			✓		✓	
Available resolution	✓	✓		✓			✓		✓	
Canvas	✓	✓		✓		✓	✓		✓	
Webgl (vendor)	*x*					*x*				
Has lied languages		✓		✓		✓	✓		✓	
Has lied OS	✓									
Touch support	✓									
Fonts	*x*									
Flash fonts	*x*	*x*	*x*	*x*	*x*	*x*	*x*		*x*	x
Plugins*	*x*	*x*		*x*		✓	*x*		*x*	
Window keys	✓	✓	✓	✓		✓	✓	✓	✓	✓
Stack trace	✓									
Web security	✓									
Popup suppression	✓					✓	✓		✓	
Document keys	✓	✓	✓	✓	✓	✓				
Navigator attributes	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
MimeTypes	*x*	*x*		*x*	✓	✓	*x*		*x*	✓
Languages	*x*	*x*		*x*		*x*	*x*		*x*	
WebSocket	✓					✓				

Table 7.5: Browser properties deviations Blink + V8

* Difference in Chromium specific browser properties as described in Table 7.2 are not being taken into consideration.
✓ = Deviation browser property.
x = Browser property could not be determined.

Standalone browser: Mozilla Firefox	Selenium + Firefox	
	Headless	Headful
Resolution	✓	
Available resolution	✓	
Webgl (vendor)	x	
Plugins	x	x
MimeTypes	x	x

Table 7.7: Browser properties deviations: Gecko + Spidermonkey

Standalone browser: Internet Explorer	Selenium+ Internet Explorer
User Agent	✓
CPU class	✓*
Navigator platform	✓*
Window keys	✓
Document keys	✓

Table 7.6: Fingerprinting properties: Trident + JScript

Standalone browser: Edge	Selenium+ Edge
Popup suppression	✓
Navigator attributes	✓

Table 7.8: Browser properties deviations: EdgeHTML + Chakra

* Is the consequence of a different CPU architecture of the web driver (Cf. appendix C).

✓ = Deviation browser property.
x = Browser property could not be determined.

The results in the tables 7.5 – 7.8 describe that the deviating browser properties are very diverse. A major reason for this diversity is the comparison of a headless browser against a standalone headful browser. A headless browser does not always contain browser properties (e.g. plugins) related to the user interface simply because it does need them.
On top of that: most web-bot driven browsers contain deviating browser properties in the following categories:

1 **Web-bot fingerprint**
Browser properties that are strongly relating and only exist when a browser instance is driven by a web bot.
The *$cdc_asdjflasutopfhvcZLmcfl_* property on the global DOM *document* object is an example of a browser property in this category.

2 **Browser fingerprint**
Browser properties that can be used to uniquely identify a browser instance. The *plugins* browser property of the global DOM *navigator* object is an example of a browser property in this category. The *plugins* property can be used to uniquely identify a browser but also to identify a web-bot driven browser instance (cf. Table 7.5 and Table 7.7).

In general, we can conclude that every web-bot browser instance contains different browsers properties compared to the standalone browsers. Below, the most remarkable deviating browser properties are described.
Appendix C gives a detailed overview of the deviating browser properties.

Deviating browsers in comparison to the used standalone browsers
The browser properties of the PhantomJS browser instance are the most deviant in comparison to the standalone Google Chrome browser properties.
We must emphasize the fact that PhantomJS is a complete other different browser than Google Chrome although both browsers share the same layout and JavaScript engine. To give an example; PhantomJS cannot be driven manually.
There are similar concerns for the NightmareJS browser instance although the NightmareJS browser can be driven manually: it is a completely different browser than Google Chrome.
For the deviating browser properties of the aforementioned web-bot browser instances this means that not all their deviating properties should be taken into consideration for the detection of web bots.
We only took into account web-bot specific properties like for instance the *userAgent* and global DOM object *Window* keys.
The *userAgent* reveals for both web-bot browser instance its origin by the literal "PhantomJS" in relation to PhantomJS and "Electron" in relation to NightmareJS.
The Window keys contain specific properties like '_phantom' and '__nightmare ' for respectively PhantomJS and NightmareJS.
On top of that fundamental browser properties that could not be determined like (*plugins* and *webgl)* can also be used to distinguish PhantomJS or NightmareJS for a standalone browser.

Headless browsers
For all headless browsers the supported MimeTypes and installed plugins could not be determined which makes a headless browser detectable based upon those properties.
On top of that, the google chrome based headless browser instances can be identified by the 'HeadlessChrome' literal in the *userAgent* property.

Selenium driven and automated Chromium browsers
The Selenium driven and automated Chromium browsers can be detected by added properties to the global DOM objects as illustrated Table 7.9.

Global DOM Object	Added key
Navigator	webdriver
Document	$cdc_asdjflasutopfhvcZLmcfl_ __webdriver_script_fn

Table 7.9: Detectable automated browser properties

Stealth web-bot driven browser instances
The headful web-bot driven browser instances do have the least deviation browser properties. Aforementioned makes sense because headful browser instances are also being used by human beings browsing the internet.
The automated Google Chrome together with the Selenium driven Firefox and Edge browsers instance are the stealthiest browser instances observed but still detectable based on global DOM object keys and / or installed plugins.

Figure 7-1 and Figure 7-2 illustrating the distribution of the browser properties deviations for respectively *headless* and *headful* web-bot browser instances.

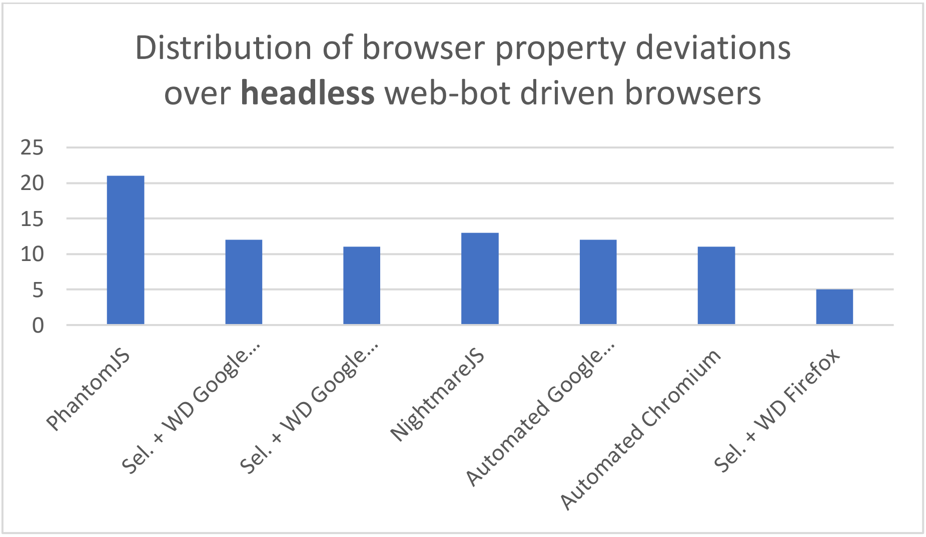


Figure 7-1: Distribution browser property deviations over headless web-bot driven browsers

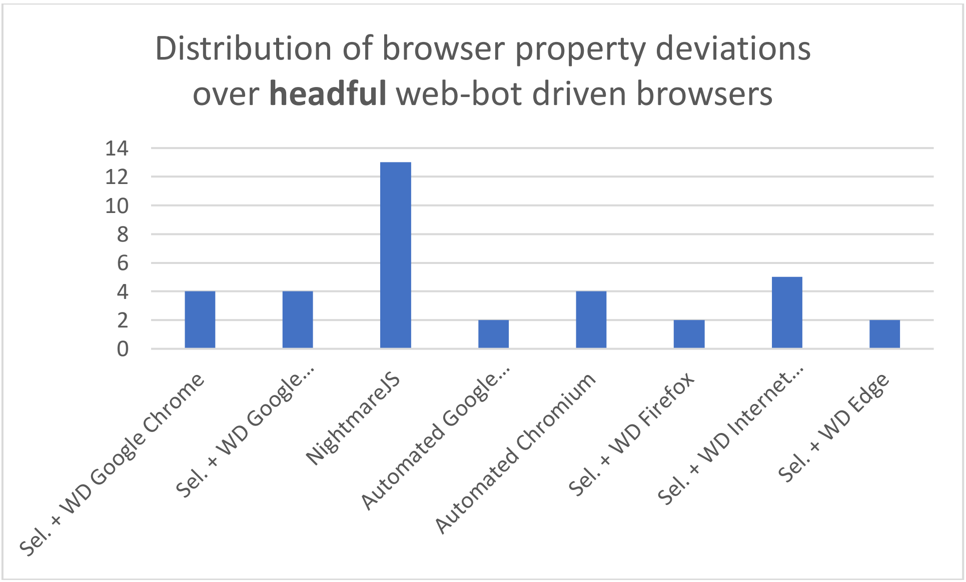


Figure 7-2: Distribution of browser property deviations over headful web-bot driven browsers

Based upon the number of deviating browser properties we can indicate how likely it is for a web-bot browser instance to get detected by a web-bot detection implementation as described in Table 7.10.

Number of deviating browser properties	Risk of getting detected
0 – 3	Low
4 - 8	Medium
> 8	High

Table 7.10: Risk classification of getting detecting by number of deviations.

7.4 Web-bot fingerprint surface

This chapter describes the transformation from deviating web-bot browser properties to a web-bot fingerprint surface that only contains the browser properties that can be used for the detection of web-bot detection.

We want to use the deviating web-bot browser properties described in paragraph 7.3 as parameters for detecting web-bot detection on the internet (cf. chapter 10).
Unfortunately, not every deviating web-bot browser property is evenly suitable for detecting web-bot detection.
To be able to detect web-bot detection based upon those browser properties we need to filter unusable browser properties and evaluate the relevance of the browser properties.

As described in paragraph Table 7.4: the deviating browser properties can be classified into two categories, respectively: *browser fingerprinting properties* and specific *web-bot fingerprinting properties* as illustrated in Figure 7-3.

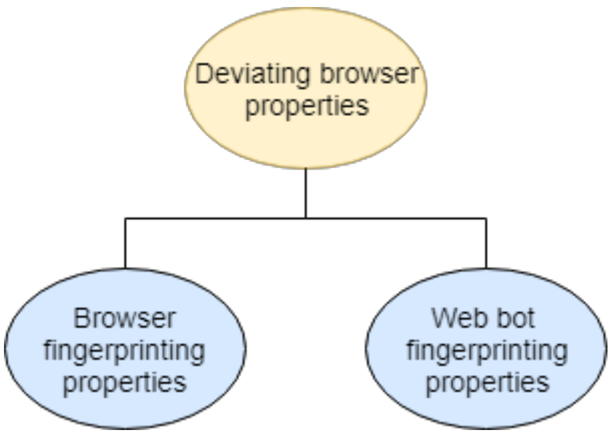

Deviating browser properties
Browser fingerprinting properties
Web bot fingerprinting properties

Figure 7-3: Deviation browser properties contains browser fingerprinting properties and web-bot fingerprinting properties

In general web-bot specific web-bot properties are stronger in the context of web-bot detection than the fingerprinting properties. A single web-bot property is specific enough to detect a web bot, this in contrast to the browser fingerprinting properties.
A browser property like for instance browser *plugins* can be used to detect a web-bot can raise a false positive as plugins might not be installed in a standalone browser.
For that reason, we decided that browser fingerprinting properties can be used to detect a web-bot but only in combination with other browser fingerprinting properties (or web-bot properties). We embodied the aforementioned construction by assigning scores to deviation browser properties and uses a threshold based on the score to determine if a website implements web-bot detection (cf. paragraph 8.1.2).
Table 7.11 describes the scores and relevance of the observed deviating browser properties (scores are relative to an empiric found threshold of 12 (cf. chapter 8)).

Browser property	Suitable for detection	Score	Category*	Remark
User Agent	Yes	12	1, 2	The values *PhantomJS*, *HeadlessChrome* and *Electron* of the browser property *userAgent* can be used to detect a web-bot driven browser instance. The WOW64 literal could also indicate a web-bot emulating a Windows OS.
Color depth	Yes	0.1	2	The PhantomJS instance only contains a different colour depth than all the other instances. We believe that the PhantomJS instance could be detected based upon this setting although we cannot argue that it is web-bot specific which is reflected in the score.
Hardware concurrency	Yes	8	2	Only in case of PhantomJS the hardware concurrency attribute *differs* from its standalone embodied by the absence of the property. We believe the *absence* of this attribute is a strong indication for a headless web-bot browser instance.
(Available) resolution	No		2	The Selenium and the automated Google Chrome and Chromium headless browser instances contain different resolution attribute properties. The resolution of the web browser is just a configuration setting that we set in our case to *800 by 600*. We could have set it for instance to *1920 by 975* which would not make this browser property deviating in respect to a standalone browser instance. Aforementioned makes this browser property usable for the detection of web-bot detection.
Canvas	Yes	1.2	2	Results prove that the canvas property is very specific for a browser instance. Especially the canvas attribute can be used to detect headless browser instances. Unfortunately, the properties of the global DOM canvas object can be system / hardware specific and therefore hard to determine if it originates from a web-bot driven instance. We believe that this property can be used to detect web-bot detection although it is not strong enough to solely detect a web bot.
Webgl (vendor)	Yes	8	2	Missing Webgl information indicates the presence of headless web-bot driven browser instance.
Has lied languages	Yes	1	2	This browser property can be used for the detection of headless web-bot browser instances. The property indicates if the language of the browser UI corresponds with sequences of the available browser languages. Because the browser property heavily relates to the available languages we chose to give it a relative low score in contrast to the *languages* browser properties.
Has lies OS	No		2	Only in case of the PhantomJS browser instance this attribute is different. The cause of this deviation originates in the fact that the *FingerprintJS* library compares the User Agent with the available plugins. The PhantomJS browser instance does not contain any

				plugins. *FingerprintJS* considers a browser instance with no plugins as a browser instance that runs on Windows. This assumption is not correct and therefore we will not take this property into consideration. Instead we will focus on the single User Agent and Plugins attributes.
Touch support	Yes	0.1	2	This browser property concerns a PhantomJS specific implementation and therefore it is not a strong browser property to use for web-bot detection.
Fonts	Yes	0.2	2	*Absence* of the installed fonts within a browser may indicate a PhantomJS browser instance. Observing missing fonts might lead to a false positive as Fonts are system specific and therefore not suitable for comparison.
Flash fonts	Yes	0.4	2	The Flash fonts attribute depends on the configured / installed browser Flash Player plugin. Modern browsers do not support the Flash Player plugin anymore due to security reasons. We encountered problems while trying to install the Flash Player plugin into our system configurations. We managed to install the Flash player only in Google Chrome and Internet Explorer. Using the Flash fonts attribute to detect web-bot detection will not be fruitful. We believe this attribute is still valuable in detecting older browsers driven by web bots as they will lack the fonts exposed through the Flash plugin.
Plugins	Yes	8	2	Absence of plugins and deviating browser plugins can strongly indicate a web-bot browser instance.
Stack trace	Yes	0.2	2	By observing the stack trace may indicate a PhantomJS web browser. Detecting web-bot detection based on this browser property is hard because there are many ways to implement the detection of a web-bot based on this browser property.
Web security	Yes	0.1	2	Only in respect to the PhantomJS browser instance the *Web Security* attribute differs from its standalone versions. The web security is a configuration property of the browser in which the level of security can be configured to protect networked data and computer systems[64]. Although we do not believe that this setting can solely detect a web-bot detection, but we believe that it can help indicate web-bot detection because web-bot security could be disabled in case of a malicious activities employed by a web bot.
Popup suppression	Yes	1	2	Can be used to detect web-bot driven web bots although it is hard the detect web-bot detection as it can be implemented in many different ways.
Windows keys	Yes	12	1, 2	Contains web-bot specific keys (cf. chapter 5.2.1).

[64] https://en.wikipedia.org/wiki/Browser_security

Document keys	Yes	12	1, 2	Contains web-bot specific keys (cf. chapter 5.2.1).
Navigator attributes	Yes	12	1, 2	Contains web-bot specific keys (cf. chapter 5.2.1).
MimeTypes	Yes	8	2	Can be used to detect different web-bot browser instances.
Languages	Yes	8	2	Can be used to detect different web-bot browser instances.
WebSocket	Yes	0.2	2	We have seen that in case of a PhantomJS and a NightmareJS web-bot browser instance the structure and sequence of the WebSocket property differs. Prior observation implies that the WebSocket browser property can be used to detect a web bot. We assigned a relative low score because the web-bot detection mechanism can be implemented in many ways which make the detection of the mechanism not that straight forward.

Table 7.11: Relevance and score of deviating browser properties

* 1 = Web-bot fingerprinting properties
2 = Browser fingerprinting properties

After the classification of the deviating browser properties we can conclude that not all the deviating browser properties can be used for web-bot detection and that we can distinguish strong and weaker browser properties in respect to the detection of web-bot detection.
Strong deviating browser properties are the web-bot fingerprinting properties that can be used for detecting a web-bot based upon *one* browser property. The weaker deviating browser properties are the browser fingerprinting properties that can be used for the detection of web-bot detection by making use of *multiple* browser properties.

The deviating properties described in Table 7.11 represent the *web-bot fingerprinting surface* which serve as parameters for measuring the adoption of the internet as will be described in chapter 8.

8 The adoption of web-bot detection on the internet

This chapter describes the measurement of the adoption of web-bot detection on the internet.
Paragraph 8.1 describes the details of the *web-bot detection scanner*[65] that has been developed for measuring the adoption of web-bot detection by making use of the web-bot fingerprint described in chapter 7.
Paragraph 8.2 evaluates the employment of the web-bot detection scanner on the Alexa top 1 million.

8.1 Design and implementation web-bot detection scanner

The web-bot detection scanner has been developed as a plugin of the OpenWPM framework (cf. appendix C). OpenWPM is a web privacy tool that is being used in many studies and described in the work of Englehardt et al. [EN16].
The motivation for using the OpenWPM framework as foundation for the web-bot detection scanner are:

- **Fast start-up time.**
  The OpenWPM framework offers sufficient tooling like web site navigation, capturing screenshots, HTTP traffic analysis and database persistence to cover the basic needs for the project described in this document.
  Aforementioned made a fast start-up of measuring web-bot detection possible.
- **Robustness and performance.**
  The OpenWPM framework provides in failover and recovering mechanisms which made crawling a large numbers of web sites of the Alexa top 1 Million feasible. On top of that, the OpenWPM framework provides in the means to execute multiple full-fledged browsers in parallel which dramatically improved the performance and with that the ability to cover a large part of sites on the Alexa top 1 million list.

The web-bot detection scanner has been developed as a plugin embodied as a command in the OpenWPM framework. A command is a coded piece of business logic that is being executed isolated from other commands. Each browser instance receives a collection of commands that sequentially are being executed.
Examples of build-in OpenWPM commands are:

- *get*
  Executes a visit to a specified URL
- *save_screenshot*
  Takes a screenshot of the current page and saves it to a specified data directory.

By making use of the command architecture of the OpenWPM framework, the web-bot detection scanner is easy interchangeable in respect to other commands.
Figure 8.1 illustrates the position of the web-bot detection scanner wrapped in a command with the name ‘detect_webbot_detection’ within the OpenWPM framework.

[65] https://github.com/GabryVlot/BotDetectionScanner

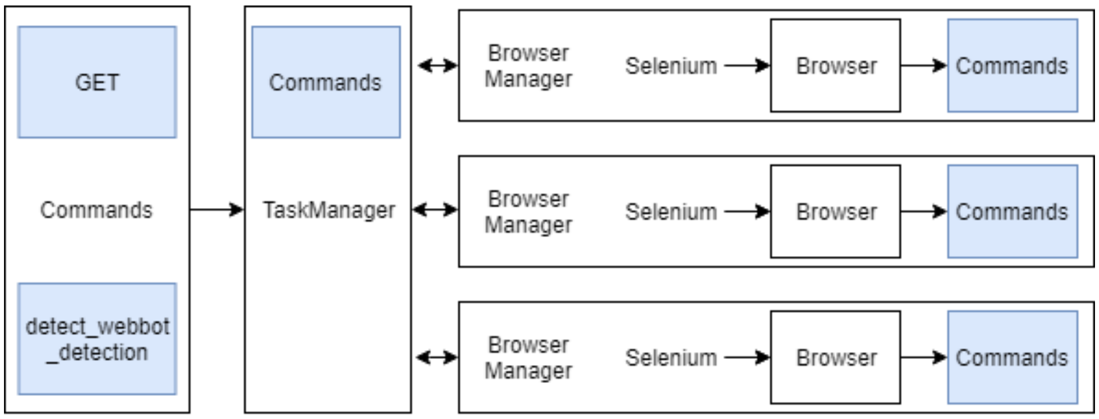


Figure 8-1: Web-bot detection scanner within the OpenWPM architecture as command

The *commands* rectangle illustrates the commands that are being executed by the browsers. In our case this is the *GET* command for opening a web page and the *detect_webbot_detection* command representing the web-bot detection scanner.
The commands are being injected into the *TaskManager* that monitors and instantiates browser managers that are responsible for starting a full fledged browser. The taskmanager iterates over the commands while injecting the Selenium driver instance which represents a browser instance.

In respect to the design of the web-bot detection scanner itself, we have strived to a modular design by separating the dynamic concepts in the application. Dynamic concepts involve mainly the *detection patterns* (cf. paragraph 8.1.2) and the *configuration settings* (e.g. database and the location of the data folder).
The configuration settings are stored in a separated file which can be easily edit without touching the innerworkings of the application.
The detection patterns are a crucial part of the application and are designed for easy extendibility and maintainability by making use of the object-oriented *class* concept in combination with an *interface*.
Aforementioned makes it easy to:

- Modify existing detection patterns
- Add new detection pattern types

8.1.1 Application flow

The main application flow of the web-bot detection scanner can be separated into four steps as illustrated in Figure 8-2

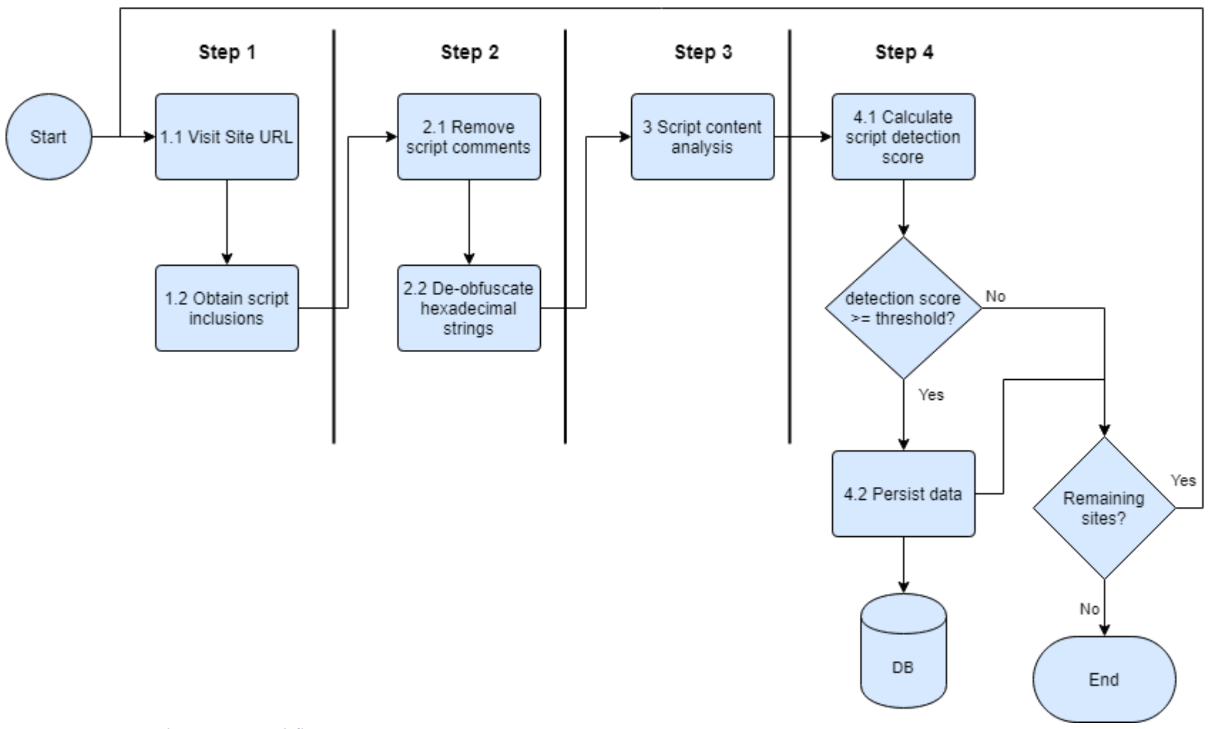

Step 1
Step 2
Step 3
Step 4
Start
1.1 Visit Site URL
1.2 Obtain script inclusions
2.1 Remove script comments
2.2 De-obfuscate hexadecimal strings
3 Script content analysis
4.1 Calculate script detection score
detection score >= threshold?
No
Yes
4.2 Persist data
DB
Remaining sites?
Yes
No
End

Figure 8-2: Application workflow

Step 1 – Visit web site and obtain script inclusions
In the first step the web-bot detection scanner inspects the homepage of a given site URL originating from the Alexa top 1 Million.
For each homepage internal script inclusions are being extracted from the homepage content and first level external script inclusions are being downloaded by their reference from the concerning webserver. If decompressing and/or decoding of the data is necessary, this also will take place in this step.

Step 2 - Script pre-processing
Subsequently all the obtained scripts are being pre-processed in step 2. The pre-processing involves:

- **Remove comments**
  Comments in scripts will never get executed and for that reason are not relevant concerning the detection of a web bot. We removed the comments to eliminate possible false positives and improves the performance of the web-bot detection (cf. paragraph 8.1.2).
- **De-obfuscating hexadecimal strings**
  The manual observation described in chapter 5 taught us that code towards web detection can be obfuscated. For that reason, we build in a mechanism that replaced all hexadecimal characters by UTF-8 characters which will make detection of literals related to web-bot detection observable.

Step 3 - Script content analysis
After pre-processing of the scripts, the contents of the scripts are being analysed on signs of web-bot detection. The analysing process makes use of four different types of *detection patterns* to detect web-bot detection in the contents of a script;
Paragraph 8.1.2 describes detection patterns more in detail.

Step 4 - Calculation script detection score and data persistence
After analysing the contents of a script on web-bot detection inclusions, the web-bot detection score of a script will be calculated based upon the detection patterns (cf. paragraph 8.1.2).
If the web-bot detection score exceeds a certain threshold, which has been determined based on an empirical observation, the script will be persisted on file and the detection patterns will be stored into the database.

8.1.2 Detection patterns and web-bot detection score calculation

Detection patterns are being used by the web-bot detection scanner to find web-bot detection inclusions in scripts and for the calculation of the detection score. Detection patterns are classified in the following types:

1. **Detector patterns**
   A detector pattern is a manually found familiar bot detection source code pattern that can be traced back to a specific *web-bot detector*. A web-bot detector is a stakeholder that is interested in the detection of a web-bot and is able to detect a web bot. Cyber security companies like for instance Distil Networks are an example of a specific *web-bot detector.*
   If a detector pattern can be found in a code inclusion on a web site the analysing process stops as we know for sure that web-bot detection takes place on this particular site.
2. **Web-bot fingerprinting patterns**
   The bot fingerprinting surface patterns are based on the bot fingerprinting surface gathered during the *determining the web-bot fingerprinting surface* phase. This type of detection pattern includes the web-bot browser fingerprinting properties classified in chapter 7.
   Also, specific web-bot fingerprinting patterns have been added that we found during the analysis of the web-bot detection implementation described in chapter 5. Due to the limitation that we used a specific software version, we did not find these patterns during the determining of the web-bot fingerprinting surface (cf. paragraph 7.4).
3. **Browser fingerprint patterns**
   The browser fingerprint patterns are based on the browser fingerprinting properties gathered during the *determining the web-bot fingerprinting surface* phase (cf. chapter 7) that can be used to detect a web-bot instance.
4. **Manually found literal patterns**
   Manually found literal patterns are common code inclusions found across different web sites that are related to web-bot detection based upon the ‘web-bot fingerprinting’ and/or ‘browser fingerprint’ patterns. The difference between *Detector patterns* and *Manually found literal patterns* is the fact that the latter cannot be traced back to a detector.
   Manually found literal patterns are only being used to categorize the existence of common code inclusion between sites that have web-bot detection implemented.
   For that reason, the manually found literal patterns are only relevant when the calculated web-bot detection score of a script exceeds the empirical found threshold of 12 (cf. section *application flow* step *4*).

Each of the aforementioned detection pattern types consist out of five properties, which are described in Table 8.1.

Property	Description
Value	The value property indicates the level of relevance in respect to web-bot detection. How higher the value, how bigger the chance that web-bot detection has been implemented on a web site. The value scale runs from 0.1 to 12 or contains the value 500. The value 500 represents web-bot detection by the detector pattern (cf. paragraph 8.1.1).
Name	Contains the name of the detection pattern which is used for categorization of detection patterns in respect to the detection value.
Patterns	The pattern property contains literals embodied in different structures that can identify web-bot detection within a script. The patterns are specific per type of detection pattern.
Prerequisites	The prerequisites property contains prerequisites that will take into account when calculating the detection score. If prerequisites are met, it will positively influence the detection score.

Determinative	The determinative property is a Boolean property which means that it can hold *True* or *False*. If the determinative property is set to True *and* the concerning detection pattern has been found in a script, the detection process will stop as we presume that the detection pattern is solely strong enough for determining web-bot detection on a web site.

Table 8.1: The properties of a detection pattern.

Detection pattern per type

The contents of a detection pattern is specific to a detection type. In this section we give an example of a detection pattern instance per type to give an impression of the usage of an instance of a detection pattern within the web-bot detection scanner.

Detector pattern

Code Snippet 8-1 illustrates an instance of the detection pattern: *detector*.
The illustrated detection pattern is able to detect a specific commercial web-bot detection code inclusion of the company PerimeterX[66]. The pattern is determinative as indicated by the last *True* parameter which means that the web-bot detection stops scanning a specific script when a detector detection pattern has been found (cf. Table 8.1).
The most interesting part is the *patterns* parameter which contains a nested array of literals. The literals *callParameter* and *window._Selenium_IDE_Recorder* are web-bot specific (cf. chapter 7) while the other parameters are implementation specific.
The literals are being used in a regular expression that is being used to search in the contents of a script file. The nested array enforces and *And* condition between the literals.
In other words; the contents of a script must contain the literal 'perimeterx' *AND* 'window._Selenium_IDE_Recorder' *AND* 'PX239' *AND* 'callPhantom'.

```
score = 500.0
perimeterX = DetectionPatternFactory.createDetectionPattern(score, 'PerimeterX', [
    ['perimeterx', 'window._Selenium_IDE_Recorder', 'PX239', 'callPhantom']
], None, True)
```

Code Snippet 8-1: Detection pattern: detector

Web-bot fingerprinting surface pattern

Code Snippet 8-2 illustrates an instance of the detection pattern: *web-bot fingerprinting surface*. The illustrated pattern is able to detect a specific web-bot characteristic; the *webdriver* property on the global DOM *navigator* object.
The pattern instance relevance is estimation to a value of 12 (first parameter) and is not determinative which means that the web-bot detection scanner does not stop after finding a web-bot detection inclusion towards the *webdriver* property within a script.
The *patterns* parameter contains a tuple with an array of literals and an OR condition. This pattern structure enforces a OR condition between the literals. The last parameter specifies the prerequisite of the pattern instance which defines that the global DOM *navigator* object must also be present in the script that is being processed.

```
navigator_webdriver = DetectionPatternFactory.createDetectionPattern(12.0, 'Webdriver', [
(["'webdriver'", '"webdriver"', '\.webdriver(?![a-zA-z-
])'], 'OR')], ['BrowserFingerprints_navigator'])
```

Code Snippet 8-2: Detection pattern: Web-bot fingerprinting surface

[66] https://www.perimeterx.com/

Browser fingerprint pattern
Code Snippet 8-3 illustrates an instance of the detection pattern: *browser fingerprint pattern*. This particular instance detects the presence of code inclusions towards available plugins on the global DOM *navigator* object of which the existence is a prerequisite for the pattern. The *patterns* parameter contains a combination between the *AND* and *OR* conditions between the literals that makes this particular instance look for *plugins.length OR window.ActiveXObject OR (plugins.length === 0 OR plugins == undefined* etcetera. The aforementioned construction will prevent that for each *plugins* occurrence within a code inclusion the score of 1.5 will be added.

```
plugins = DetectionPatternFactory.createDetectionPattern(1.5, 'Plugins', [
'plugins.length', 'window.ActiveXObject', ([['plugins.length == 0'],
 ["plugins.length === 0"], ["plugins == undefined"],["plugins === undefined"]], 'OR'),
 'x-pnacl', 'Shockwave Flash', 'ShockwaveFlash.ShockwaveFlash'
], ['BrowserFingerprints_navigator'])
```

Code Snippet 8-3: Detection pattern: Browser fingerprint pattern

Manually found literal pattern
Code Snippet 8-4 illustrates an instance of the detection pattern; *manually found literal*. Manually found literal patterns do not have a positive score assigned to them because they are only being used to detect familiar code inclusions across script files.
The pattern instance illustrated in Code Snippet 8-4 is able to detect the presence of the *FingerprintJS* library that we also have being used to obtain the web-bot fingerprinting surface (cf. chapter 7).

```
fingerprintjs = DetectionPatternFactory.createDetectionPattern(0.0, 'FingerprintJS',
['webglKey', 'hasLiedResolutionKey', 'hasLiedBrowserKey', 'hasLiedOsKey'])
```

Code Snippet 8-4: Detection pattern: Manually found literal pattern

Web-bot detection score calculation
Web-bot detection can be implemented in one script or distributed across multiple scripts in the context of a web page as illustrated in Figure 8-3.

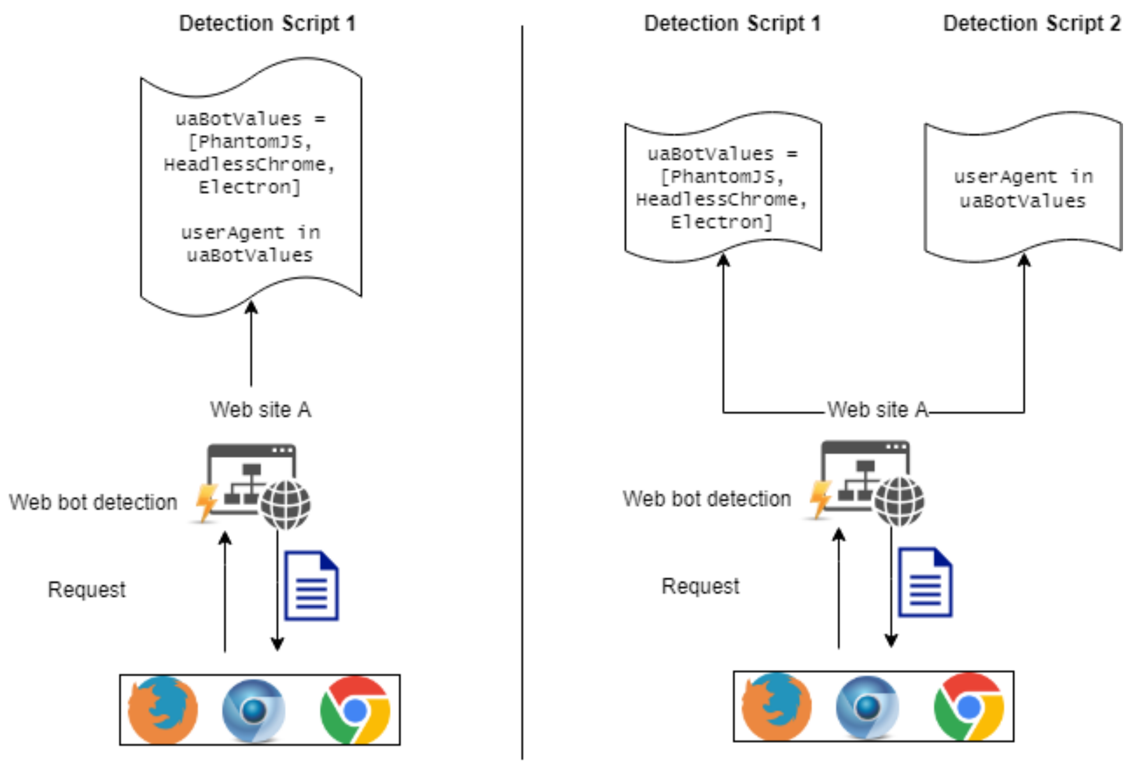


Figure 8-3: Implementation detection in one file (left) or distributed over multiple files (right)

The left situation in Figure 8-3 represents web-bot detection implemented in one script. The right situation represents web-bot detection distributed over two scripts where detection is only possible when the scripts are being used together.
In respect to the calculation of the detection score we have chosen to calculate the score *per* script because of the following two reasons:

1. **It limits the number of false positives.**
   Apparent distributed detection implementations will not be taken into consideration if the scripts are not being used in conjunction. For instance, if *detection script 1* and *detection script 2,* displayed in Figure 8-3 on the right side, are never being used in conjunction this can been seen as a false positive.
2. **The approach is in line with third party scripts**.
   The third-party scripts (e.g. script originating from FingerprintJS, Distil Networks, PerimeterX) towards web-bot detection that we observed during this study all implement their detection mechanisms in one script file. We believe that the main reason for this is the ease of deployment for providing those scripts to their customers (the web site hosts).

Distributed web-bot detection can be taken into account despite the decision to calculate the score per script. If a distributed script will be marked as web-bot detection script depends on the implementation of the separated scripts itself.
To explain aforementioned more in detail we need to take a closer look at the calculation of the score.

As described; the calculation of the detection value depends on the found detection patterns in a script and will be calculated based on the following algorithm:

FS	$=$ Collection found *web-bot fingerprint patterns*
BF	$=$ Collection found *browser fingerprint patterns*
ES	$=$ External scripts
IS	$=$ Inline scripts

```
DP = {FS, BF}
S = {ES, IS}

d = found detector pattern

for each s ∈ S:
    if d:
        return detectionScore = 500
    for each p ∈ DP:
        score = p.value * p.occurences
        if p in FS:
            if not prerequisites(p):
                score = score / 2
    return detectionScore += score
```

First the collection of found detection patterns will be checked for the presence of a *detector pattern*. If a *detector pattern* is present the detection score is 500.
If there is no *detector pattern* present, the application checks for *web-bot fingerprinting* and *browser fingerprint* patterns.
The *web-bot fingerprinting* patterns contain a higher relevance value in respect to the *browser fingerprint* patterns because it is *most likely* that they are being used for web-bot detection. *Browser fingerprint* patterns indicate that the script *can* be used to detect a web bot.
For both patterns the detection score will be calculated by multiplying the amount of occurrence of a pattern by its score.
In respect to the *web-bot finger printing surface* the prerequisite plays a role in the calculation of the score. If a prerequisite does not satisfy, for instance, if the *useragent* global DOM navigator object (which is a prerequisite) has not been found in the script in the distributed web-bot detection example provided in Figure 8-3, the total score will be devided by two.
Aforementioned rule has been introduced to eliminate false positives on one hand and to support distributed web-bot detection on the other hand.

- **False positive mitigation**
  We observed some false positives when *web-bot fingerprinting* patterns (e.g. "PhantomJS") had been found by the scanner but no relating Browser property (e.g. *navigator.userAgent*). We choose to divide the score by two which led to a score just below the empirical found threshold so the script would not be marked as web-bot detection solely based upon this pattern.
- **Distributed web-bot detection.**
  If two or more *web-bot fingerprinting* patterns would be found without passing the prerequisites the web-bot detection score will be higher than the threshold value based upon the chosen pattern relevance values. This means the distributed web-bot detection implementation will be observed by the web-bot detection scanner.
  Aforementioned approach supports a distributed web-bot detection implementation as we have seen on DollsKill.com (https://www.dollskill.com/) where all the *web-bot fingerprinting* patterns are implemented in a separated script file and the web detection that uses the patterns in another file.

8.1.3 Validation

The validation of the correctness of the web-bot detection scanner concerned an iterative process in relation with the development of the scanner.
In first place we took the downloaded files obtained during the analysis phase (cf. chapter 5) as input for the scanner. An advantage of using downloaded files was the development speed but even more important; we would not target extensively one particular web site as it might would jeopardize the

quality of the data returning from the web site host, because there was a risk involved that we could get blocked.
After the development of the scanner by making use of downloaded files we started to targeted the customer web sites of CyberSecurity companies. We found them purely by reading promotional material on the sites of the CyberSecurity companies. Aforementioned approach did not only validate the scanner but also provided us with additional *detector* and *manual found literal* patterns.

8.2 Evaluation

We employed the web-bot detection scanner to analyse the script inclusions on the web sites of the Alexa 1-million-list.
Two separate systems were used that respectively used eight and seven browsers in parallel within a time window of fourteen days. The first system analysed web sites by starting from the top of the list, where the second system started from the bottom of the list. We visited in total 431.340 sites of which 391.527 sites the script inclusions could be successful analysed of which 50.205 sites contained web-bot fingerprinting surface properties as described in Table 8.2. Future work is necessary to determine the reason why it was not possible to analyse the script inclusions of all visited sites.

Visit Sites	Analysed sites	Sites with web-bot fingerprinting surface properties	Highest score (excluding detector patterns)	Lowest score
431.340	391.527	50.205 (12,82 %)	288.4	12

Table 8.2: Sites containing web-bot fingerprinting surface property script inclusions

For the measurement of web-bot detection on a web site we took only the score of the script inclusion with the highest score.
The web site with the highest web-bot detection score (detector patterns not included as the score will be fixed to a value of 500) contained a score of 288.4. The lowest score within the results gathered concerns obviously the value of the determined threshold which is 12. A web site with a web-bot detection score of 12 equals a web site that at least contains a web-bot fingerprint pattern.

Obviously, a web site can contain multiple scripts that have web-bot detection implemented. In other words: scripts that have a web-bot detection value of 12 or higher.
We believe that we must take a look at all the scripts that have a web-bot detection value higher than 12 to get a better understanding of script coverage in respect to the adoption of web-bot detection.
For that reason, we want to take a closer look at the details of the analysed scripts that are key for the adoption of web-bot detection on the internet.
The details of the scripts that contain web-bot fingerprinting details are described in Table 8.3.

Total script inclusions	Duplicates	Unique scripts
60.103	45.125 (75,07%)	14.978 (24.92%)

Table 8.3: Scripts details containing web-bot fingerprinting properties

In general, we have found a lot of duplicated scripts inclusions. Main reason is the distribution of generic modules / libraries like for instance *WebFont.js*: a NodeJS module for generation of fonts or *FingerPrintJS* (cf. paragraph 7.2) or *OneSignal* a standardised module for the implementation of push notifications.

The coverage per web-bot detection level for the *web-bot detection sites* (the web sites that include scripts containing web-bot detection code inclusions) and all script inclusions (including the duplicates) are described in respectively Table 8.4 and Table 8.5.

We can classify the level of web-bot detection in a script by using the detection patterns:

- **Detector patterns / Web-bot fingerprinting patterns**
  Web-bot detection takes place.
- **Browser fingerprints**
  Browser fingerprint takes place that can also identify a web-bot.

Sites with web-bot fingerprinting surface properties	Detector patterns	Web-bot fingerprinting patterns	Browser fingerprinting patterns
50.205	1.467 (2.92 %)	44.592 (88.81 %)	4.146 (8.25%)

Table 8.4: Web-bot detection level coverage per detection pattern (web-bot detection sites)

Total script inclusions	Detector patterns	Web-bot fingerprinting patterns	Browser fingerprinting patterns
60.103	1.490 (0,74 %)	53.633 (89,83 %)	58.612 (97,52 %) Without the 'web-bot fingerprinting surface' patterns: 4.980 (8.28 %)

Table 8.5: Web-bot detection level coverage per detection pattern (all script inclusions)

By far most scripts contained web-bot fingerprinting patterns which also always included browser fingerprints as can be concluded from the column 'Browser fingerprints' in Table 8.5. The column 'browser fingerprints' is a category represented in the majority of scripts. The reason for it is the co-existence with the 'web-bot fingerprinting patterns.
We observed 4.146 / 4.980 scripts with only 'browser fingerprints' patterns which are the weakest in relation to the observation of web-bot detection.

The reason for the relatively low percentage of 'detector patterns' is that we only used eight detector patterns to detect web-bot detectors as described in Table 8.6. Possible future work towards analysing the web-bot detection scripts will further improve the coverage of this pattern.

Detector pattern	Occurrence
Distil Networks	1.114 (74.77%)
PerimeterX	229 (15.31 %)
DataDome	53 (3.6 %)
AdScore	41 (2.7%)
PerfDrive	53 (3.6%)

Table 8.6: Script occurrences per detector pattern

Web-bot detection score domains
The web-bot detection score gives an indication how likely it is for a script to detect a web bot. How higher the score; the more likely a script can detect a web bot.
We found that each detection pattern has got its own domain. For instance, the domain for the detector pattern is the 500 score. The score domain for 'web-bot fingerprinting pattern' and 'browser fingerprint pattern' are respectively: 12 - 288.4 and 12 - 24.5.
Figure 8-4 illustrates the score domains and score distribution across the observed sites.

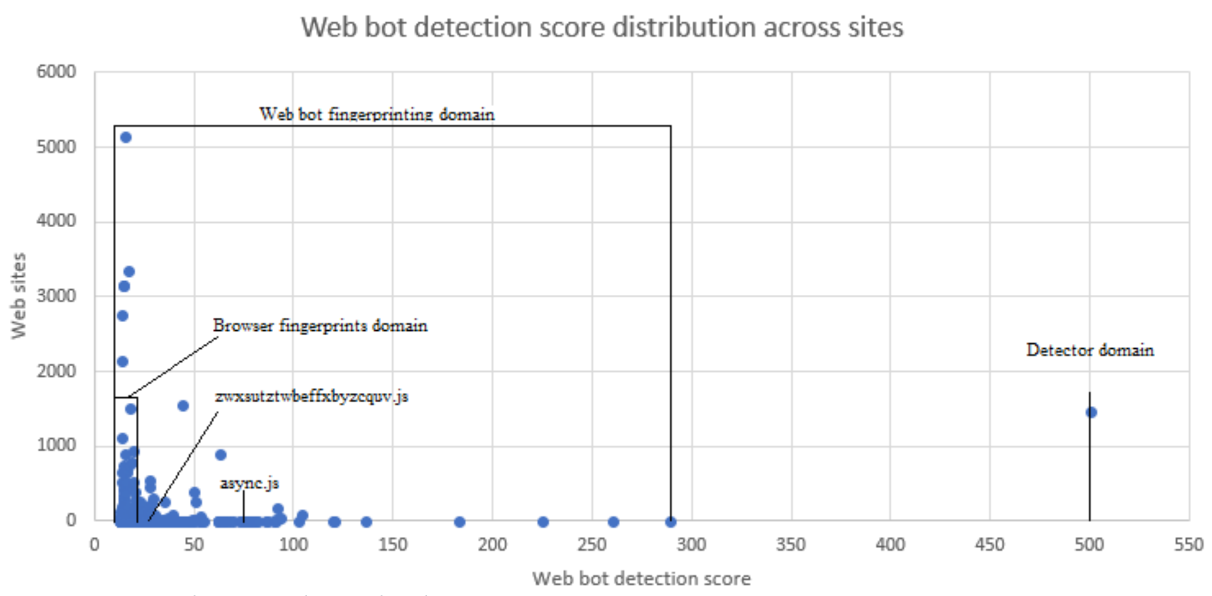


Figure 8-4: Score domains and score distribution across sites

Figure 8-4 illustrates that the density of web-bot detection values within sites is higher in the range between 12 and 30. In general in this region reside web-bot detection script that detect one or two detectable web bots. The density within the browser fingerprints domain can be clarified through heavy distribution of external modules.
The scripts discussed in chapter 5 can be detected by the web-bot detection scanner based on its detector pattern (Distil Networks) which will be assigned the value 500.
To illustrate that the found scripts do not concern edge cases we have put the scripts with their web-bot detection score based upon their web-bot fingerprint surface properties in the graph.

8.3 Improvements

The current implementation of the web-bot detection scanner leaves room for improvement concerning antipatterns and honeypot detection.

- **Antipatterns**
  Antipatterns can improve the detection of web-bot detection by specifying patterns that seem to be web-bot detection specific but not really are. By using antipatterns, false positives could be eliminated and even the speed of the web-bot detection scanner could be improved by discarding script containing an antipattern.
- **Honeypot detection**
  Honeypots[67] (for instance invisible form elements) can be used to detect a web bot. Currently the web-bot detection scanner does not have any patterns in place for detecting honeypots. This means that purely detection of web bots based on honeypot elements will slip through our web-bot detection mechanism.
  An improvement embodied by extending the web-bot detection patterns would be to include honeypot detection patterns that are able to detect elements that will not get visible in the contents of a particular web page.

[67] https://en.wikipedia.org/wiki/Honeypot_(computing)

8.4 Limitations and risk

We believe that the results described in chapter 8.2 are just the tip of the iceberg given the limitations of the web-bot detection scanner which are:

- **Homepage visits**
  The scanner visits only the homepages of the sites in the Alexa top 1 million list. We chose for this approach as it was the most pragmatic approach in relation to a relative short amount of time.
  We believe that the most non-login web sites that implement web-bot detection are employing web-bot detection on their homepage. In respect to web sites that require to login; it makes sense to have web-bot detection on the deeper level web pages in the 'secured zone'.
- **First level plugin analysis**
  Currently only first level scripts are being analysed for web-bot detection inclusions. This means that when a non-web-bot detection script contains another script that does contains web-bot detection this will not be detected by scanner.
  Although recursive script analysis is easy to implement, we chose for a one level detection implementation from the perspective of performance and available data extraction time.
- **Static code analysis**
  The static code analysis implemented by the web-bot detection scanner (cf. paragraph 8.1) is not able to detect heavy obfuscated code inclusions. Code Snippet 8-5 illustrates a web-bot detection pattern that cannot be detected by the web-bot detection scanner. The reason for it is because the literal *'_callPhantom'* (which is a web-bot fingerprinting surface pattern) is being dynamically constructed during compile time.
  Observation of the illustrated web-bot detection pattern is not impossible but difficult as it would require backtracking the variables / constants used and interpret them in the right way which indeed would involve an interpreter.
  We decided to not support such a complex mechanism as we find the chance that it does exists very small. Heavy obfuscated as illustrated is very hard to maintain for a web-bot detector.

```
const obfuscatedArray = ['_', 'c', 'a', 'l', 'l', 'P', 'h', 'a', 'n', 't', 'o', 'm'];

if (window[obfuscatedArray.join('')])
    console.log('It is a Bot!!!!');
```

Code Snippet 8-5: Heavy obfuscated web-bot detection pattern

9 Observing deviations

In this chapter we describe the observation of deviations on web sites that implement web-bot detection. The sites that we used originating from the *measurement of the adoption of web-bot detection* phase described in chapter 8.

Web-bot detection can lead to deviations on a web page and both are therefore closely related to each other.
In the light of studies that are using a web bot; deviations on a web page can harm the outcome of a study (cf. chapter 1) or might be not of relevance in the context of a particular study.
For instance, when a study towards linked in profiles will not be able to extract advertisements from a web page this will probably not be relevant contrary to studies towards malvertisement[68].
Malvertisement is the distribution of malware via vulnerable and corrupted advertisements.

To get more insight in the consequences of web-bot detection in respect to the employment of web bots in studies, we manually observed web sites for different types of deviations as described in paragraph 9.1. Paragraph 9.2 describes the results of the manual observation.

9.1 Approach

We used 20 web sites that contain web-bot detection mechanisms as we observed during the *adoption of web-bot detection* phase (cf. chapter 8). From all the *web-bot detection sites* we chose the web sites with the highest web-bot detection score in respect to detection pattern and score diversity.

- **Detection pattern diversity**
  In respect to the detector pattern we avoided the use of duplicate detector patterns.
- **Deviation score diversity**
  In respect to the deviation score we avoided the use of deviations scores.

By applying the aforementioned diversities, we strived to find different types of deviations. The web sites used, together with their score are described in Table 9.1.

Site	Score	Detection pattern
http://aetnabetterhealth.com	500	Detector pattern (Distil Networks)
http://allergyandair.com	500	Detector pattern (PerimeterX)
http://yify-films.com	500	Detector pattern (AdScore)
http://diabetesincontrol.com	500	Detector pattern (Distil CDN)
http://eiendomspriser.no	500	Detector pattern (PerfDrive)
http://lequotidiendupharmacien.fr	500	Detector pattern (DataDome)
http://proredirector.com	288.4	Web-bot fingerprinting / browser fingerprinting pattern
http://mmobeast.com	260.29	Web-bot fingerprinting / browser fingerprinting pattern
http://diapasonmag.fr	224.6	Web-bot fingerprinting / browser fingerprinting pattern
http://moutech.com	183	Web-bot fingerprinting / browser fingerprinting pattern
http://zip.net	136.1	Web-bot fingerprinting / browser fingerprinting pattern
http://cine974.com	120.5	Web-bot fingerprinting / browser fingerprinting pattern
http://wtftattoos.com	119.5	Web-bot fingerprinting / browser fingerprinting pattern
http://amdelplata.com	103.5	Web-bot fingerprinting / browser fingerprinting pattern
http://kiyu.tw	102.1	Web-bot fingerprinting / browser fingerprinting pattern
http://anpfiff.info	92.7	Web-bot fingerprinting / browser fingerprinting pattern
http://321kochen.tv	91.7	Web-bot fingerprinting / browser fingerprinting pattern
http://affairesdegars.com	90.9	Web-bot fingerprinting / browser fingerprinting pattern
http://bellechic.com	89.7	Web-bot fingerprinting / browser fingerprinting pattern

[68] https://en.wikipedia.org/wiki/Malvertising

http://pc-4you.ru	86.7	Web-bot fingerprinting / browser fingerprinting pattern

Table 9.1: Web-bot detection sites used for the observation of different types of deviations

We observed deviations by manually comparing the screenshots of the homepage of the sites described in Table 9.1 between a visit of a web-bot driven and a standalone browser instance.
We used PhantomJS as web-bot driven browser instance and Google Chrome a standalone web browser instance to visits the web sites each from a different machine and IP address. Especially the fact that each machine will have its own IP address is important because the standalone browse could not be confronted with web-bot detection deviations caused by the web-bot browser instance.
We chose PhantomJS because it is the most detectable web-bot browser instance which increased the chance to find different types of deviations.
We visited each web site five times to ensure that there were enough occasions for the server host to detect the web-bot and respond with a deviation.

9.2 Different types of deviations

By making use of screenshots we observed deviations in the visual representation (cf. paragraph 2.1.2). We found deviations in 17 sites that can be classified in the types *blocked response*, *blocked web page content* and *deviation in the content of the webpage* and are described in Table 9.2. The first column in Table 9.2 describes the deviation type followed by the column that aggregates the occurrences of combinations of deviations. The last column lists the sites per deviation combinations. Details of the deviations types *blocked web page content* and *deviation in the content of the web page* can be found in appendix E.

Deviation type	Deviation(s)	Site(s)
Blocked response	Main page not loaded	http://amdelplata.com
Blocked content	Blocked by Captcha in the first visit.	http://aetnabetterhealth.com http://lequotidiendupharmacien.fr http://allergyandair.com
	Blocked by message.	http://eiendomspriser.no
Deviation in the content of the web page.	No(t all the) advertisements could be loaded.	http://diabetesincontrol.com http://zip.net http://wtftattoos.com http://affairesdegars.com http://anpfiff.info http://pc-4you.ru
	- Different page layout. - No all the advertisements could be loaded.	http://proredirector.com
	- No videos where loaded. - No all the advertisements could be loaded.	http://diapasonmag.fr http://321kochen.tv
	Not all videos could be loaded.	http://cine974.com
	Login dialog not displayed.	http://kiyu.tw
	Difference in page layout.	http://bellechic.com

Table 9.2: Observed web page deviation types

Below we will describe the details of the web-bot browser instance deviations regarding the different categories as described in Table 9.2.

Blocked response
On the site *http://amdelplata.com* we have observed that the main page could not be load after displaying of a ‘splash screen’ (a web page containing only a logo).
This behaviour can be identified as a blocked response where the response (the main page) is not being send back to the client as cause of web-bot detection (cf. paragraph 2.1.2).

Blocked content
On four web pages we observed blocked content in three of them by making use of a Captcha while visiting the web page for the first time.
On the web site http://eiendomspriser.no we observed blocking embodied by full page message in Norwegian describing that our IP address was temporary blocked for this site.

Deviation in the content of the web page
During our observation we found four types of deviations:
- Videos could not be loaded.
- Difference in page layout.
- Advertisements could not be loaded.
- Login dialog not displayed.

We believe that the deviations *Videos could not be loaded* and *difference in page layout* are not caused by web-bot detection but simply because we utilized a headless browser instance. As described in chapter 7: a headless browser as PhantomJS does not have plugins installed. For this reason, videos cannot be loaded that require a particular plugin.
We also blame the difference in page layout to the use of a headless browser where the HTML rendering is slightly different in respect to the Google Chrome browser. We observed that all the information is available on the web pages but only the alignment is a bit off.
We do believe that lack of advertisements on the different web pages has been caused by web-bot detection. In first place this observation is in line with the observation in the work of Englehardt et al. (cf. chapter 1). Secondly it is plausible that certain content delivery networks (CDNs) or advertisers do not serve advertisements to web-bot simply because advertisements are not relevant for web bots from the perspective of an advertiser. In fact, serving advertisements to web bots increases the risk that fraudulent revenues are being generated [Whi16].
We also believe that the absence of the login dialog on *http://kiyu.tw* has been caused by web-bot detection. In the line of hiding advertisements for web bots it makes sense that also web site hiding access to authorized content.

Overall, we state that the deviations listed in Table 9.2 are most likely caused by web-bot detection because:
- We observed web-bot detection client side code (cf. chapter 8).
- We believe that the observed deviations cannot be caused by excessive network traffic. The relevant web pages have been visited to a maximum of three till the observation of the deviation appeared.

We can conclude that we can observe different deviation types based upon web-bot detection where the observations type *deviation in the content of the web page* is the most harmful in relation to a study. The reason for it is because a deviation is harder to observe, contrary to blocked content.
We claim that a malvertisement study in ‘the wild’ by making use of a PhantomJS web-bot browser instance will lead to unreliable results if the results are not being mitigated in respect to web-bot detection.

10 Conclusion, discussion and future work

In this document we described a research towards the adoption of web-bot detection on the internet and its relation to web-bot based studies.
We proposed a six-phase methodology that provided us with the means to measure web-bot detection on the internet and to observe web page deviations caused by web-bot detection.

By conducting the individual phases of the proposed methodology, we answered the following research questions:

RQ1: How to proof and measure the existence of web-bot detection?
A manual observation supplemented by an investigation of the innerworkings of a specific web site with a strong web-bot detection smell, proved the existence of web-bot detection.
We observed deviations when visiting the web page by making use of a web-bot driven browser instance in contrast to a visit by a human being using a regular browser.
We substantiated the presence of a web-bot detection implementation by an investigation of the inner workings which revealed a client side implementation of a web-bot detection mechanism.
We validated the web-bot detection mechanism by constructing a countermeasure that bypassed the web-bot detection mechanism. After implementing the countermeasure no deviations have observed.
Investigation of the innerworkings revealed that the web-bot detection mechanism is able to detect web bots client side based upon the properties exposed by a browser which makes web-bot detection measurable.

RQ2: To what extent is web-bot detection adopted by web sites on the internet?
We have developed a web-bot detection scanner that we have used to measure the adoption of web-bot detection. The web-bot detection scanner visited 43.130 sites of the Alexa top 1-million-list of which 391.527 sites could be used for analysing of the script inclusions. Future work is necessary to determine the reason why it was not possible to analyse the script inclusions of all visited sites.
The web sites have been analysed for web-bot detection code inclusions in scripts by making use of a so-called *web-bot fingerprint surface*. The web-bot fingerprint surface exists out of browser properties that can be used to detect a web bot.
The web-bot fingerprint surface has been implemented in the web-bot detection scanner as detection patterns. Detection patterns have been used to assign a web-bot detection score per script inclusion and classify a script into one or more web detection levels that indicates the ability to detect a web-bot (cf. paragraph 8.2).
Both the detection patterns *Detector pattern* and *Web-bot fingerprinting pattern* indicate that web-bot detection takes place. The detection pattern *browser fingerprinting pattern* indicates that browser fingerprinting takes place that is also able to detect a web bot.
The results of the employment of the web-bot detection scanner points out that 12.82% of the investigated sites are able to detect a web-bot browser instance.
The coverage per detection pattern is described in Table 10.1.

Visit sites	Analysed sites	Detector pattern	Web-bot fingerprinting pattern	Browser fingerprinting pattern
431.340	391.527 (100%)	1.467 (2.92 %)	44.592 (88.81 %)	4.146 (8.25%)

Table 10.1: Web-bot detection level coverage (web-bot detection sites)

An adoption rate over the 12% is a big number especially considered that we only scratched the surface. Regarding the measurement of the adoption of web-bot detection on the internet, we only took the homepages and first level scripts of the web sites in the Alexa 1-million-list into account.
Furthermore, we are aware that there exist other ways to detect a web-bot that we did not take into account like for instance honeypot elements.

RQ3: Can we distinguish different types of deviations based upon web-bot detection?
We observed three different types of deviations by manually comparing the screenshots of web sites visited with a (PhantomJS) web-bot driven and a standalone browser instance. A deviation is an observable difference on a web page caused by the employment of a web-bot in comparison to the web page visited by a human being.
For the comparison we selected 20 sites that contained web-bot detection inclusions based on their web-bot detection score and diversity of implemented detection patterns (cf. RQ2).
In 16 of the 20 sites we found deviations that can trace back to web-bot detection classified into one of the following types:

- **Blocked response**
  Web site decides to send no response to the client.
- **Blocked content**
  The content of the web page has been blocked by a CAPTCHA.
- **Deviation in the content of the web page**
  Content contains a deviation in the visual representation of a web page.

We observed deviations on web-bot detection sites by making use of PhantomJS which in general resulted in deviations in the interactive content of a web page. Based upon the aforementioned facts, we claim that the results of a study towards malvertisement in the wild by making uses of PhantomJS for acquiring the data, cannot be realistic.
We believe that the observed deviations caused by web-bot detection are just the tip of the iceberg simply because more subtle deviations can be easily implemented by server hosts which are hard to detect.

RQ4: How likely is it for web-bot based studies to get detected by web-bot detection?
The likeliness of getting detected depends on the used data extraction technique. In the context of the research described, the data extraction technique equals the used web-bot framework to automate a browser instance.
During our research we investigated nine web-bot frameworks / browser combinations to find deviating browser properties of web-bot browser instances in respect to a standalone (human driven) browser instance. The risk of getting detected per web-bot framework / browser combination is described in Table 10.2 established by making use of the web-bot detection level described in paragraph 7.3.

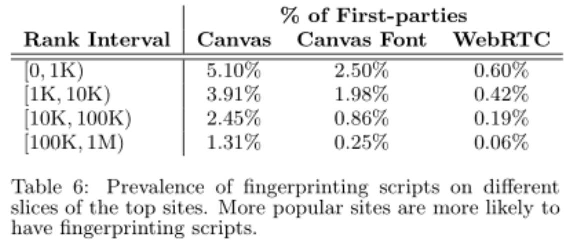
Web-bot driven browser instance	Headless	Headful
PhantomJS	High	/
Selenium + Google Chrome	High	Medium
Selenium + Google Chromium	High	Medium
NightmareJS	High	High
Automated Google Chrome	High	Low
Automated Chromium	High	Medium
Selenium + Firefox	Medium	Low
Selenium + Internet explorer	/	Medium
Selenium + Edge	/	Low

Table 10.2: Risk of getting detected per web-bot browser instance.

In general, we found that each web-bot driven browser instance contains at least two deviating browser properties that can be used to detect a web-bot driven browser instance. Headless browsers contain the most deviating browser properties.
A browser instance driven by PhantomJS contains by far the most deviating browser properties (21) which makes it the most detectable browser instance.

The browser instances Edge or Firefox driven by Selenium or an automated Chromium browser instance are the stealthiest, all with two deviating browser properties.
PhantomJS is in our opinion too detectable for targeting a large number of unfamiliar sites on the internet. Therefore, we would advise against the use of PhantomJS for data extraction in the wild. PhantomJS could still be an added value towards specific web-bot based studies.
Besides the data extraction techniques is the used data extraction tactics of influence on the likeliness of getting detected by a web-bot detection implementation. During our research we visited web sites ‘in the wild’ by making use of one web bot. In this case a web site will be visited on its most one time. During a manual observation of a web-bot detection implementation (cf. chapter 5) we have seen that one visit might be not enough to trigger web-bot detection.
If we target a web site many times it is more likely that web-bot detection will be triggered (and probably also triggers rate limiting which is beyond the scope of our research) as there will be more interactions between client and server which increases the chance of getting detected.

We argue that web-bot detection is a real threat for web-bot based studies. A web-bot detection rate of 12.82% is a large number, especially seen in relation to the results of some of the described web-bot based studies described in paragraph 2.4. The work of Nikiforakis et al. examined 10.000 web sites of the Alexa top 1-milion list and found 40 sites (0.4%) that utilizing fingerprinting code from three commercial providers [NKJ+13]. Web-bot detection could thwart the results of the work of Nikiforakis et al. by applying deviations on a large number of the web pages of the sites with fingerprinting code.
In ratio with the measured web-bot detection adoption rate, 33 web pages of the 10.000 sites could contain web-bot detection (worst-case scenario). Even without the relative uncertain *browser fingerprinting patterns* script inclusions (cf. Table 10.1) the adoption rate comes down to 11.76% which still can have a great influence on the results of the study. Nikiforakis et al. did not mentioned the used web-bot techniques in their paper. Concerning the popularity of the web-bot techniques used during our research it is very likely that they employed a web-bot that is detectable by web-bot detection (cf. Table 10.1).
Nikiforakis et al. did not mentioned the risk of getting detected by web-bot detection in their paper and therefore we assume that web-bot detection has not been taken into consideration which makes it a real threat for their study.
Similar applies to the work of Englehardt et al. that applied a PhantomJS and Selenium + Firefox web-bot driven browser instances to find fingerprinting script inclusions [EN16]. Both web-bot browser instances are detectable by web-bot detection within a level of respectively High and Medium / low. Also, Engelhardt et al. did also not mention the possible effect of web-bot detection on the results of their study. The results described in their paper as illustrated in Figure 10-1 indicate that web-bot detection can have a big influence on the results of the study. Illustrates the coverage of possible web-bot detection sites per rank interval that can be related to the sites containing fingerprinting scripts in Figure 10-1.



Rank Interval	% of First-parties		
	Canvas	Canvas Font	WebRTC
[0, 1K)	5.10%	2.50%	0.60%
[1K, 10K)	3.91%	1.98%	0.42%
[10K, 100K)	2.45%	0.86%	0.19%
[100K, 1M)	1.31%	0.25%	0.06%

Table 6: Prevalence of fingerprinting scripts on different slices of the top sites. More popular sites are more likely to have fingerprinting scripts.

Figure 10-1: Adoption of fingerprinting scripts on top sites [EN16]

#Sites relative to the rank interval illustrated in Figure 10-1.	Web-bot detection adoption rate
1.000	0.03%
10.000	0.33%
100.000	3.27%
1.000.000	32.74%

Table 10.3: Web-bot detection adoption rates per rank interval (worst case scenario).

10.1 Discussion

We described that it is plausible that the results of web-bot based studies are being thwarted by web-bot detection. Web-bot detection will be triggered by web-bot specific properties exposed by the browser.
Scientists need web bots to elicit knowledge from the wealth of information available on the internet as manual extraction is infeasible. What can be done to make the relationship between web-bot detection web sites and scientist workable?
The prior-mentioned questions can be approached from the perspective of a web site, software manufacturer / open source community and of an end user (a scientist).

Web site
In our opinion should web-bot detection not indiscriminately take place based on the properties of a browser which equals an automated software product. Web-bot detection must focus on wrong usage of a web site. For instance, excessive traffic generation or fraudulent login attempts.
Unfortunately, this is web site specific and therefore out of the hands of the end user.
Software manufacturer / open source community
Software manufacturers / open source community can contribute to a better employability of web bots by scientist by a change in the automated software tools.
Web-bot detection takes place on the properties exposed by a browser which can be divided in deviating browser properties and web-bot browser properties. If a web-bot detection browser instance does not want to get detected it needs to make sure that it does not stand out in comparison to a main stream standalone browser instance.
Prior-mentioned is the responsibility of the software manufacturer / open source community of the automated software. For PhantomJS this is quite challenging as it contains 21 deviating browser properties. If the open source community decides to let PhantomJS stay under the web-bot detection 'radar' it will need to change the value of the userAgent, change their error stack trace so that it is in line with the stack trace of a standalone browser instance, change color depth etcetera.
Concerning other automated software like for instance the Chromium web browser there is less work involved. The Chromium based browser contains a property *webdriver* which indicates that the browser is being driven by a web bot. Removing this property would make the web-bot driven Chromium browser less detectable.
Although technical modifications are possible to make a web-bot driven browser instance more stealth, we do not believe software manufactures are willing to cooperate simply because it involves a too high risk for their product. Can you imagine what will happen to the Chromium product if a DDOS attack can only be performed with an automated version of Chromium?
There might be also an ethical aspect involved concerning the relation between software manufacturer / open source community and web site that states that an automated browser must be always recognizable.

In general, we observed the most deviating browser properties in respect to headless browser instances. The major reason for this is that the headless browser instances do not contain display related properties like plugins, fonts, MimeTypes and languages.
One can argue that the gap between headless and headful browser instances will be bridged by the rich feature set provided by the HTML 5 language.
For plugins this might be the case as less and less plugins (e.g. the Flash plugin) are required nowadays to experience an interactive web site.
We do not believe that the evolution of the HTML language will overcome the deviations of the other mentioned properties.
Bridging the gap between headless and headful browser instances is another responsibility for the software manufacturer / open source community as headless browsers will stay popular in respect to their performance and scalability in large scale research studies.

End user
When web-bot detection still takes place on the properties exposed by a browser and the employment of a web-bot still leads to deviating browser properties, then only the end user can make the web-bot undetectable.
We would opt for taking a web-bot browser instance with the least amount of deviating browser properties and spoof the deviating properties. We believe that a Selenium driven Edge browser instance is the least detectable web-bot browser instance as it concerns a relative new browser with only two deviating browser properties.

10.2 Future work

The research area of web-bot detection turned out to be a rich research area where we foresee the following fruitful future works:

- The research described only scratched the surface of web-bot detection on the internet. Future research towards other web-bot detection implementations is necessary to provide a more complete measurement of the adoption of web-bot detection on the internet.
  For now, we think of the following two additional web-bot detection mechanisms:
  - Honeypots
  - Google reCAPTCHA[69] (Google reCAPTCHA seems not to be browser instance specific)
- In future work the accuracy of the web-bot detection scanner can be optimized by:
  - Analysing the observed web-bot detection sites for familiar detector patterns. The detector detection pattern can be extended with these patterns which makes the scanner more accurate as it is able to detect proven web-bot detection implementation. Also, a broader coverage of detector pattern helps to get an overview of notorious web-bot detection detectors (companies) on the internet.
  - Introducing anti-patterns. Antipatterns can improve the detection of web-bot detection by specifying patterns that seem to be web-bot detection specific but not really are. By using antipatterns, false positives could be eliminated and even the speed of the web-bot detection scanner could be improved by discarding script containing an antipattern.
- Integrating the web-bot detection scanner in data extraction frameworks (e.g. OpenWPM). Integration can help scientist mitigate the results of web-bot detection sites. We can image that it will be very fruitful to know if web-bot detection takes place when extracting data from a particular site. Feedback provided by the web-bot detection scanner can be used to mitigate the results or even to avoid web-bot detection.
- Web-bot detection and the determining of deviations are very close related. Unfortunately, the described study indicates that web-bot detection *can* take place. To decide if web-bot detection takes place we also need to observe deviations on a web page. Determination of deviations is a hard thing to do but we believe it becomes more feasible if web-bot detection will be integrated in a future project towards the detection of deviations. Web-bot detection indicate when a deviation *can* occur which could trigger deviation detection that checks the structural and visual representations for deviations.
- Impact analyse of web-bot detection by conducting a repeatable study. This researched described that the results of a study can be thwarted by web-bot detection. A study must be repeated to prove the relationship between web-bot detection and the quality of the results of a web-bot based study.

[69] https://www.google.com/recaptcha/intro/v3beta.html

Appendices

A. Taxonomy of automatic data extraction methods

A taxonomy of automatic data extraction methods can be constructed on different kind of levels. Figure A1 illustrates an overview of automatic data extraction levels considered from a top down approach: process management, approach, strategy, extractable content and supported technology.

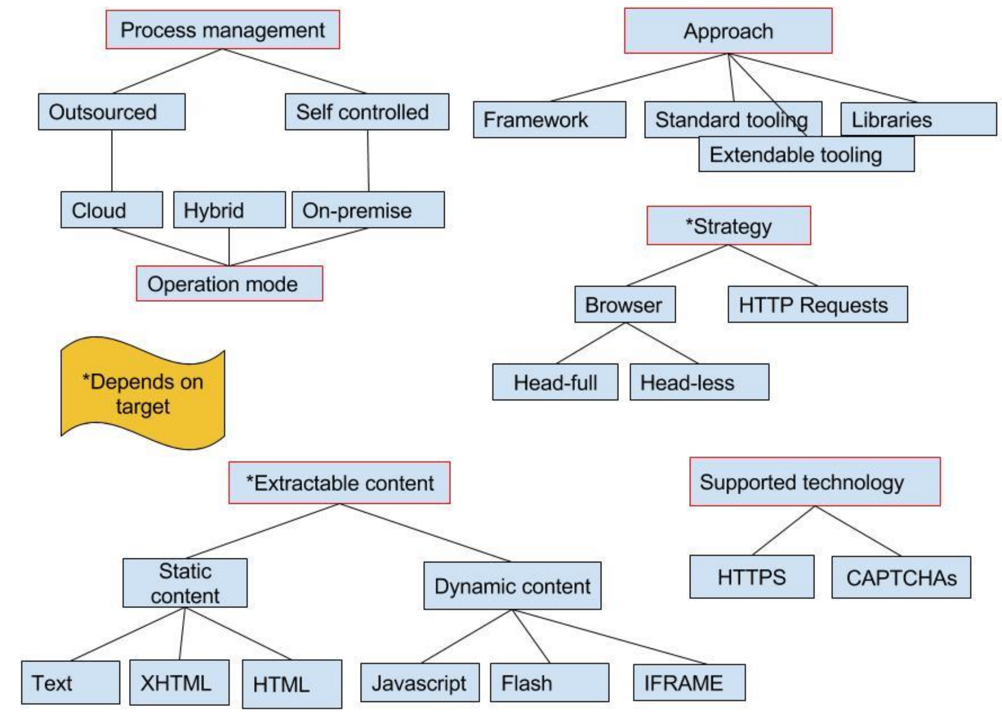


Figure A 1: Overview of automatic data extraction levels

The next paragraphs give an overview of a possible taxonomy on each level with taken into account the pro's and con's.

A.1 Management of the data extraction process

At the highest level we can divide automatic data extraction in the way the data extraction process is managed. Basically, this will leave us with two types of data extraction; data extraction can be offered as a service by a third party company or can be performed under own supervision.

There are commercial companies, like Prompt Cloud[70], that are offering data extraction from the web as a service. For a certain kind amount money, the data extraction process can be outsourced to this external party. Big advantage is that no knowledge of data extraction techniques is needed and it will cost (hardly) extra company resources in terms of time. In most cases you are dealing with data extraction specialists.
There are also disadvantages; what happens to the data for what is paid for? Maybe it will also be sold to other companies. Besides there is little to no control over the configuration of the data extraction process. Big question is; how custom-made is the data extraction process in relation to the needs of the customer?

[70] https://www.promptcloud.com/scraping-service/

When data extraction is performed under own control it is clear what is going to happen with the extracted data. Also, in most cases the data extraction method can be tailored to the needs of the data request. On the other side, data extraction is an expertise and it will take effort and company resources to gain and maintain the knowledge.

Method	Pros	Cons
As a service	● No (deep) data extraction knowledge required. ● No need for extra company resources.	● Uncertain what will happen to ‘your’ data. ● Uncertain what will happen with fingerprinting of the data extraction behaviour. ● No detailed control on the configuration.
Self controlled	● Extracted data is self controlled ● Self-control over exposing fingerprinting behaviour. ● Full control concerning what to extract (configuration)	● Need for data extraction knowledge. ● Need for data extraction company resources

Table A 1: Pros and cons process management

A.2 Approach

The work of Glez-Pena et al. describes that an automated data extraction implementation can be divided into four flavours; by making use of libraries, a framework, by making use of a standard ‘out of the box’ tool or using an extendable standard tool [GLL+14].

Using a library (like Wget or libcurl) for automatic data extraction has the great advantage of writing a tool within a familiar language and language structure. In general a library provides a solid base for creating a tool but are not that versatile. For covering a wide range of functionalities libraries with different skills have to be combined.

A framework provides the building blocks for making a data extraction tool. In this perspective a tool can be seen as a specialism of a framework.
By making use of a framework (like Scrapy (http://scrapy.org) for instance) a tool can be extended very easily by writing additional code. In that way it is possible to tailor the data extraction implementation in detail to the desired needs. Another big additional advantage is that a framework in most cases is been driven by an underlying community. If the community is large it can provide in an already developed functionality. Of course, the power of it depends on the size and activity of this community. Another advantage is that it is (partly) free of charge (open source driven).
Disadvantages: usability (no out of the box wizards) and the need for specific knowledge. For instance, implementing a data extraction solution with Scrapy, programming knowledge (Python) is necessary.

Most out of the box tools are in general configurable but not extendable. This means that in this case the tool selections must exactly suit the needs of the data extraction purposes. In case of facing limitations, there is no cure available. On the other hand; in most cases a lot of data extraction knowledge is implemented in those tools. Only configuration knowledge is required to start data extraction.

There are a lot of existing tools that can be used ‘out of the box’ and provide the ability to extend its functionality for example by writing scripts. A big advantage is the start-up time. It is not necessary to write a bunch of code to get going. When you are facing limitations, you might be able to solve it with writing extensions. By now you probably need some programming skills, but you are able to make it suit to the needs of your business.
Disadvantage; compared to a solution written on top of a framework, it tends to have a more restrictive character.

Approach	Pros	Cons
Libraries	● The ability to create an automatic data extraction tool in familiar programming language.	● Several libraries need to be integrated (i.e. one for webaccess and one for parsing)
Framework	● More integrated solution compared to libraries. ● Extendable to fit the required needs . ● Re-use of already developed blocks/functionality by the community. ● Knowledge sharing through the community. ● (In most cases) free of charge.	● Lack of ‘out of the box’ functionality. ● Specific (programming) knowledge required
Standard tooling	● No profound knowledge required. ● Startup time, easy to get going.	● Not extendable, only configurable. ● Not very versatile. Usually it concerns a specialism. ● Commercial distribution
Extendable standard tooling	● Start-up time, easy to get going. ● Provides the opportunity to make adjustments for tailoring it to the specific demands.	● Restrictive character concerning the flexibility of a data extraction framework ● Commercial distribution

Table A 2: Pros and cons concerning different approaches

A.3 Operation mode

The location of the execution of data extraction can be distinguished in on-premise (also server farm included), cloud or a hybrid situation between these two.

In an on-premise mode the data extraction software runs standalone. No interference of other processes are involved. Advantage of this situation is basically what you see is what you get. The data extraction user knows what happens with the data. Disadvantage; amount of firepower available. The software has only local resources available.

When data extraction is performed in the cloud we are talking about providing data extraction as a service. In this case this does not concern outsourcing of the data extraction process (as mentioned in paragraph A.1) but basically running a data extraction tool remote (for example; Import.io https://www.import.io). Running the data extraction process as a service has the main advantage that worrying about system resources and maintaining them is not necessary. This configuration will probably also contribute to scalability. Big disadvantage is the unknown of the origin of the data extracted and the big question is; what happens to 'my' data?

In a hybrid situation we are talking about a local running process in combination with interaction of the internet. Basically we are talking about a distributive situation. In a distributive situation the local data extraction software is able to spin up remote cloud based data extraction tasks (https://nutch.apache.org)

Mode	Pros	Cons
On-premise	● Transparent, tactile. It's easier to see what happens to the extracted data.	● Limited resources
Cloud	● Not the burden of managing resources. ● Scalability ● Availability	● Not able to guarantee what will happen with the extracted data.
Hybrid	See "Cloud"	See "Cloud"

Table A 3: Pros and cons concerning different operation modes

A.4 data extraction strategy

In relation to web data extraction there are different strategies possible. Each one of them has their own pros and cons. This paragraph will discuss the following strategies:

- Browser
- Http request

Browser

A popular data extraction strategy is to use a testing scripting framework for crawling and data extraction web pages. In fact, we are dealing with programmable browsers. Browser based data extraction comes in two flavours; a headless (like http://phantomjs.org/) and a head-full browser (like http://www.seleniumhq.org/projects/webdriver/) implementation. A headless implementation does not spin up a browser window but just acts on the api's of the underlying webpage renderer. A head-full implementation does spin up a browser window.
A big advantage of a headless implementation is that it can run in a continuous integration environment. Some say the performance it better in contrary to the head-full brother. A disadvantage is clearly the lack of transparency because you don't see what is going on.

Basically, this strategy mimics the behaviour of an end user with all its advantages. This strategy will cause to execute dynamic content like JavaScript that also can be extracted. Another advantage is that is hard to determine if it is a robot or a real human being access the web page (genuine cookie and http header behaviour).
Of course, there are also disadvantages; in respect to non-browser strategies seems browser based data extraction very heavy on the resources. Also, it is said that data extraction can easily be detected by analytics tools (i.e. Google analytics).

HTTP Request

Another approach is to make use of http request / responses. This approach will deal with HTTP requests on a lower level, basically mimic (some) of the functionality of a browser. There are plenty

tools that are using this strategy. Main advantage is the performance contrary to the browser strategy. There are also concerns. For instance how realistic are the results that are extracted by making use of the HTTP approach. In other words; can these results also be obtained by an end user (with a browser)? Besides it is harder to extract data from dynamic content. It first needs to be executed by the data extraction tool (if this is even possible).
When you choose to create your own data extraction tool based on HTTP requests this might be a steep learning curve. Bare in mind that it is not only about HTTP request but also about site rendering, executing site JavaScript etcetera.

Strategy	Pros	Cons
Browser - Headless	● Mimics end user behaviour and therefore realistic concerning the real world. ● Out of the box execution of dynamic content. ● Behaviour is hard to distinct from a robot. ● Runs in a continuous integration environment. ● Better performance taken into account a head-full browser.	● In respect to non browser approach; it will need a lot of resources. ● Heavy data extraction can be more easily determined (page visited). ● Execution is not transparent. This makes development and debugging a bit harder compared to a head-full browser.
Browser - headful	● Mimics end user behaviour ● Out of the box execution of dynamic content. ● Behaviour is hard to distinct from a robot. ● Execution is transparent. Makes development and bug finding easier compared to a headless browser.	● Does not run in a continuous integration environment. ● Worse performance taken into account a headless browser.
Http Requests	● Performance	● Realism of the extracted data. ● Data extraction for dynamic content needs additional effort. When developing an data extraction tool based on this strategy the following concerns need to taken into the consideration: ● Need for a good understanding of HTTP requests / responses in combination with other browser specific implementation techniques. ● Additional boilerplate code for mimic browser behaviour.

Table A 4: Pros and cons concerning different data extraction strategies

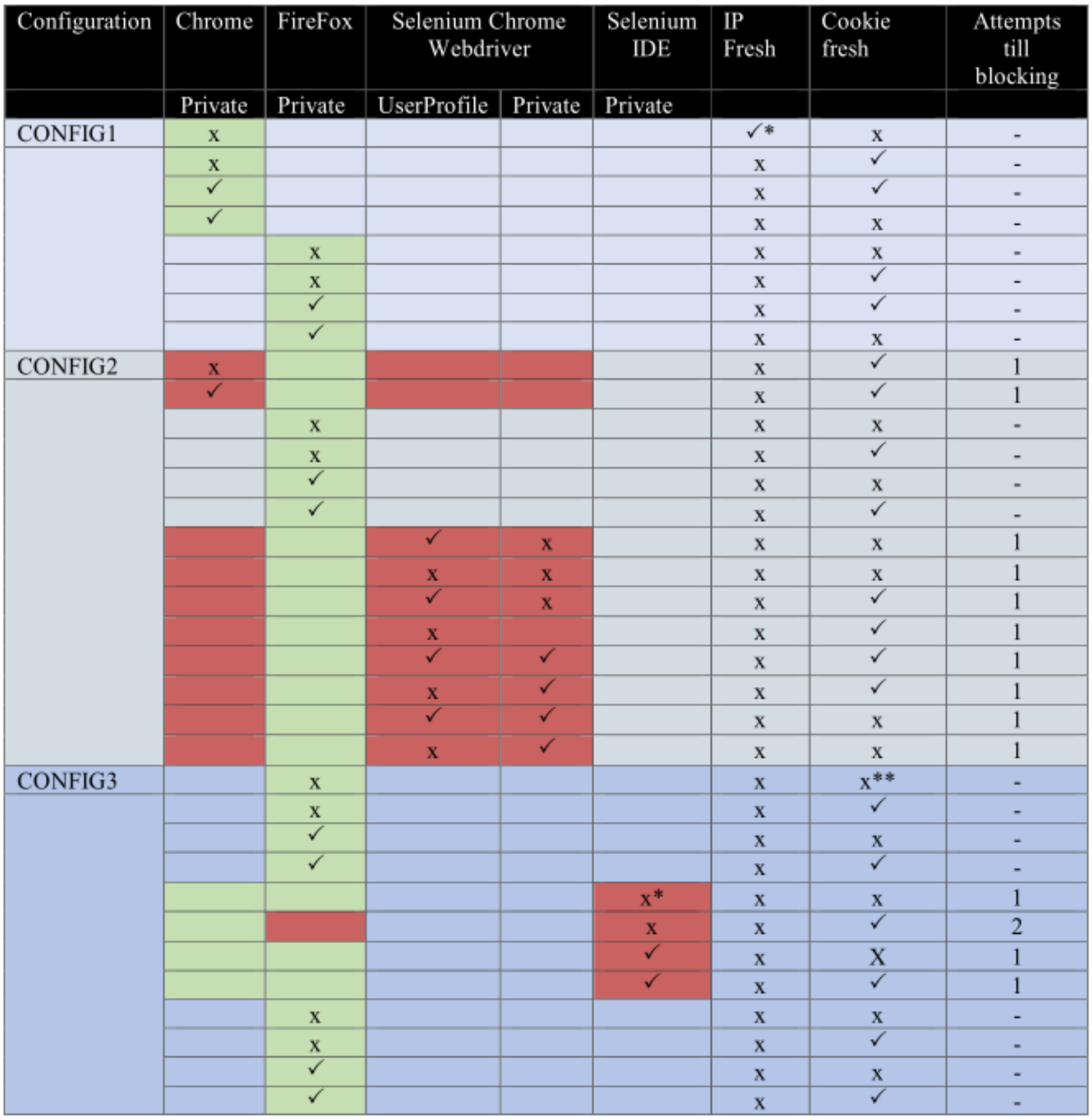
A.5 Extractable content

When describing a taxonomy of automatic data extraction methods, probably the most important one is the amount of support concerning different types of content. There are automatic data extraction tools with only support for static content as well support for static and dynamic content. Static content can be just plain text or XHTML or HTML. Nowadays the most popular type of dynamic content is JavaScript. But also, the content of FLASH and IFRAME can be fruitful to extract. The combination of product type and chosen strategy are mainly responsible for the supported types of content.
This paragraph doesn't give an overview of the pros and cons in relation to the specific extractable content types basically because they depend on the intention of scrapping.

A.6 Supported technology

The supported technology determines the employability of an automatic data extraction tool. This paragraph does not give an inexhaustible list of technologies. Two mainstream technologies that must be taken into consideration are, HTTPS and CAPTCHAs. Support for HTTPS will give the tool the ability to extract data on private sites, of course this will increase the scope enormously. An automatic extraction tool aims for gathering as much data as possible. An automatic extraction tool could be blocked by CAPTCHAs if the tool does not provide support for it.

B. Details manual observation web-bot detection on StubHub.com



Configuration	Chrome	FireFox	Selenium Chrome Webdriver		Selenium IDE	IP Fresh	Cookie fresh	Attempts till blocking
	Private	Private	UserProfile	Private	Private			
CONFIG1	🟩 x					✓*	x	-
	🟩 x					x	✓	-
	🟩 ✓					x	✓	-
	🟩 ✓					x	x	-
		🟩 x				x	x	-
		🟩 x				x	✓	-
		🟩 ✓				x	✓	-
		🟩 ✓				x	x	-
CONFIG2	🟥 x	🟩	🟥	🟥		x	✓	1
	🟥 ✓	🟩	🟥	🟥		x	✓	1
		🟩 x				x	x	-
		🟩 x				x	✓	-
		🟩 ✓				x	x	-
		🟩 ✓				x	✓	-
	🟥	🟩	🟥 ✓	🟥 x		x	x	1
	🟥	🟩	🟥 x	🟥 x		x	x	1
	🟥	🟩	🟥 ✓	🟥 x		x	✓	1
	🟥	🟩	🟥 x	🟥		x	✓	1
	🟥	🟩	🟥 ✓	🟥 ✓		x	✓	1
	🟥	🟩	🟥 x	🟥 ✓		x	✓	1
	🟥	🟩	🟥 ✓	🟥 ✓		x	x	1
	🟥	🟩	🟥 x	🟥 ✓		x	x	1
CONFIG3		🟩 x				x	x**	-
		🟩 x				x	✓	-
		🟩 ✓				x	x	-
		🟩 ✓				x	✓	-
	🟩	🟩			🟥 x*	x	x	1
	🟩	🟥			🟥 x	x	✓	2
	🟩	🟩			🟥 ✓	x	X	1
	🟩	🟩			🟥 ✓	x	✓	1
		🟩 x				x	x	-
		🟩 x				x	✓	-
		🟩 ✓				x	x	-
		🟩 ✓				x	✓	-

Figure B 2: Details manual observation web-bot detection on StubHub.com

🟩	Normal working
🟥	Blocked web page content

*IP address has not been used for visiting StubHub.com for over more than a week.
**New VirtualBox image. First visit no cookies where present.

[CONFIG1]: No automated software installed (Windows 10). Google Chrome and Firefox installed.
[CONFIG2]: Selenium with Chrome WebDriver (Linux Ubuntu). Google Chrome and Firefox installed.
[CONFIG3]: Selenium IDE Firefox plugin. Firefox and Google Chrome installed

C. Deviating browser properties

In this appendix we describe the deviating browser properties that are the result of a comparison between standalone browser instances and a web-bot driven browser instances (cf. chapter 7.3). We describe the deviating browser properties per browser family classification (cf. chapter 6).

C.1 Blink + V8

This paragraph described the deviating browser property of web-bot driven browser instances compared against a standalone Google Chrome browser instance.

PhantomJS

userAgent

The userAgent string of the PhantomJS browser instance (illustrated Code Snippet C- 1) differs from the Google Chrome implementations on three aspects:
it cannot determine the Unix graphical system (x-window[71]), indicated as *unknown*.
The userAgent contains the literal *PhantomJS/2.1.1* (which represents the browser and version).
The *WebKit* version that is slightly older than the Google Chrome browser illustrated in Code Snippet C- 2.

```
Mozilla / 5.0(Unknown; Linux x86_64) AppleWebKit / 538.1(KHTML, like Gecko) PhantomJS / 2
.1.1 Safari / 538.1
```

Code Snippet C- 1: userAgent PhantomJS

```
Mozilla / 5.0(X11; Linux x86_64) AppleWebKit / 537.36(KHTML, like Gecko) Chrome / 64.0.32
82.140 Safari / 537.36
```

Code Snippet C- 2: userAgent Google Chrome

Canvas

The rendering of the *canvas* DOM element differs in many ways in comparison of the *canvas* DOM element rendered by the Google Chrome browser instance.

Windows keys

Missing	Added
origin	webSecurity
customElements	missingBind
external	stackTrace
onanimationend	autoClosePopup
onanimationiteration	offscreenBuffering
onanimationstart	event
isSecureContext	webkitIndexedDB
oncancel	ontouchmove
onclose	ontouchstart
oncuechange	webkitNotifications
ontoggle	ontouchend
onwheel	console
onauxclick	ontouchcancel
ongotpointercapture	_phantom
onlostpointercapture	callPhantom
onpointerdown	getMatchedCSSRules
onpointermove	showModalDialog
onpointerup	webkitConvertPointFrom
onpointercancel	NodeToPage
onpointerover	webkitConvertPointFrom
onpointerout	PageToNode
onpointerenter	

[71] https://www.computerhope.com/jargon/x/xwin.htm

onpointerleave	webkitCancelRequestAnimationFrame
onafterprint	
onbeforeprint	
onlanguagechange	
onmessageerror	
onrejectionhandled	
onunhandledrejection	
requestIdleCallback	
cancelIdleCallback	
createImageBitmap	
onappinstalled	
onbeforeinstallprompt	
caches	
ondevicemotion	
ondeviceorientation	
ondeviceorientationabsolute	
webkitStorageInfo	
fetch	
visualViewport	
speechSynthesis	
webkitRequestFileSystem	
webkitResolveLocalFileSystemURL	
chrome	
attr	
TEMPORARY	
PERSISTENT	

Table C- 1: Deviating window keys (PhantomJS browser instance)

Stack trace

The PhantomJS stack trace stands out from the other browser stack traces that are identical:

```
stackTraces@http://10.0.2.2:8080/index.html:124:21
global code@ http: //10.0.2.2:8080/index.html:7:39
```

Code Snippet C- 3: PhantomJS error stack trace

```
"TypeError: Cannot read property '0' of null
at stackTraces(http: //10.0.2.2:8080/index.html:124:21)
        at http: //10.0.2.2:8080/index.html:7:28"
```

Code Snippet C- 4: PhantomJS Google Chrome error stack trace

Document keys

Missing	Added
origin	releaseCapture
xmlEncoding	mozSetImageElement
xmlVersion	mozCancelFullScreen
xmlStandalone	enableStyleSheetsForSet
selectedStylesheetSet	caretPositionFromPoint
preferredStylesheetSet	onbeforescriptexecute
webkitVisibilityState	onafterscriptexecute
webkitHidden	mozFullScreen
onbeforecopy	mozFullScreenEnabled
onbeforecut	mozFullScreenElement
onbeforepaste	selectedStyleSheetSet
onsearch	lastStyleSheetSet
oncancel	preferredStyleSheetSet
oncuechange	styleSheetSets

onmousewheel	ondragexit
webkitIsFullScreen	onloadend
webkitCurrentFullScreenElement	onshow
webkitFullscreenEnabled	onmozfullscreenchange
webkitFullscreenElement	onmozfullscreenerror
onwebkitfullscreenchange	onanimationcancel
onwebkitfullscreenerror	onanimationend
registerElement	onanimationiteration
caretRangeFromPoint	onanimationstart
webkitCancelFullScreen	ontransitioncancel
webkitExitFullscreen	ontransitionend
	ontransitionrun
	ontransitionstart
	onwebkitanimationend
	onwebkitanimationiteration
	onwebkitanimationstart
	onwebkittransitionend

Table C- 2: Deviating document keys (PhantomJS browser instance)

Navigator attributes

Missing	Added	Modified
maxTouchPoints		vendor
hardwareConcurrency		appVersion
doNotTrack		userAgent
geolocation		
mediaDevices		
connection		
webkitTemporaryStorage		
webkitPersistentStorage		
serviceWorker		
budget		
permissions		
presentation		

Table C- 3: Deviating navigator attributes (PhantomJS)

Websocket

```
readyState:0,onclose:null,bufferedAmount:0,extensions:,onerror:null,onopen:null,url:ws://
localhost:8080,binaryType:blob,protocol:,URL:ws://localhost:8080,onmessage:null,CLOSED:3,
CLOSING:2,CONNECTING:0,send:function send() {
    [native code]
},dispatchEvent:function dispatchEvent() {
    [native code]
},addEventListener:function addEventListener() {
    [native code]
},OPEN:1,close:function close() {
    [native code]
},removeEventListener:function removeEventListener() {
    [native code]
}
```

Code Snippet C- 5:Deviating WebSocket structure (PhantomJS browser instance)

```
CONNECTING:0,OPEN:1,CLOSING:2,CLOSED:3,url:ws://localhost:8080/,readyState:0,bufferedAmou
nt:0,onopen:null,onerror:null,onclose:null,extensions:,protocol:,onmessage:null,binaryTyp
e:blob,close:function close() { [native code] },send:function send() { [native code]
},addEventListener:function addEventListener() { [native code]
},removeEventListener:function removeEventListener() { [native code]
},dispatchEvent:function dispatchEvent() { [native code] }
```

Code Snippet C- 6: Structure WebSocket object Google Chrome browser instance)

Single browser property deviations

Browser property	Standalone driven browser property	Web-bot driven browser property value
Color depth	32	24
Resolution	1920x975	1855x951
Available resolution	1855x951	1024x768
Has lied OS*	False	True
Touch support**	0, False, False	0, True, True
Web security	True	False
Popup suppression	False	True

Table C- 4: Single browser property deviations (PhantomJS browser instance)

* *FingerprintJS* library compares the User Agent with the available plugins. The PhantomJS browser instance does not contain any plugins. *FingerprintJS* considers a browser instance with no plugins as a browser instance that runs on Windows. This assumption is not correct and therefore we will not take this property into consideration. Instead we will focus on the single User Agent and Plugins attributes.
** First parameter represents the *maxproperties* property of the global DOM *navigator* object. The second parameter represent respectively the ability of creation of the *TouchEvent* on the global DOM *document* object and the presence of the *ontouchstart* property on the global DOM *window* object.

Selenium Driven Browser instances

User agent

When using Selenium to drive Google Chrome in headless mode, the user agent contains the literal '*HeadlessChrome'* contrary to the user agent of Google Chrome described in Code Snippet C- 2.

Canvas

The rendering of the *canvas* DOM element differs in many ways in comparison of the *canvas* DOM element rendered by the Google Chrome browser instance.

Window keys

window_keys	Missing
Selenium Chrome (Headless)	chrome attr
Selenium Chrome (Headful)	attr
Selenium Chromium (Headles)	chrome

Table C- 5: Deviating window keys (Selenium driven browser instance)

Document keys

document_keys	Added
SeleniumChrome (HL and HF) SeleniumChromium	$cdc_asdjflasutopfhvcZLmcfl_

Table C- 6: Deviating document keys (Selenium driven browser instance)

Navigator attributes

navigator	Added	Modified
Selenium Chome (HL) Selenium Chromium (HL)	webdriver	appVersion userAgent
Selenium Chome (HF) Selenium Chromium (HF)	webdriver	

Table C- 7: Deviating document keys (Selenium driven browser instances)

MimeTypes

```
application / pdf, application / x - google - chrome - pdf, application / x - ppapi - wid
evine - cdm
```

Code Snippet C- 7: Deviating MimeTypes (Chromium headful browser instance)

```
application / futuresplash, application / pdf, application / x - google - chrome - pdf, a
pplication / x - nacl, application / x - pnacl, application / x - ppapi - widevine - cdm,
 application / x - shockwave - flash
```

Code Snippet C- 8: MimeTypes Google Chrome browser instance

Single browser property deviations

Browser property	Standalone driven browser property	Web-bot driven browser property value
Resolution	1920x975	640x480
Available resolution	1855x951	640x480
Has lied languages*	False	True
Popup suppression	False	True

Table C- 8: Single browser property deviations (Selenium driven browser instances)

* The property indicates if the language of the browser UI corresponds with sequences of the available browser languages.

NightmareJS

User agent

The user agent of the NightmareJS browser instance contains the literal *'Electron'*.

Canvas

The rendering of the *canvas* DOM element differs in many ways in comparison of the *canvas* DOM element rendered by the Google Chrome browser instance.

Plugins

```
Chromium PDF Viewer::Portable Document Format::application / x - google - chrome –
pdf~pdf
```

Code Snippet C- 9: Detectable plugins NightmareJS browser instance

```
Chrome PDF Plugin::Portable Document Format::application / x - google - chrome - pdf~pdf,
 Chrome PDF Viewer::::application / pdf~pdf, Native Client::::application / x - nacl~, ap
plication / x - pnacl~, Shockwave Flash::Shockwave Flash 29.0 r0::application / x - shock
wave - flash~swf, application / futuresplash~spl, Widevine Content Decryption Module::Ena
bles Widevine licenses
for playback of HTML audio / video content.(version: 1.4.9.1070)::application / x - ppapi
 - widevine - cdm~
```

Code Snippet C- 10: Detectable plugins Google Chrome browser instance

Window keys

Missing	Added
onafterprint	getMatchedCSSRules
onbeforeprint	console
onmessageerror	__nightmare
onappinstalled	onshow

onbeforeinstallprompt visualViewport chrome attr	

Table C- 9: Deviating window keys (NightmareJS browser instance)

Document keys

Missing	Added
onvisibilitychange	onshow

Table C- 10: Deviating document keys (NightmareJS browser instance)

Navigator keys

Missing	Added	Modified
connection budget	credentials	appVersion userAgent

Table C- 11: Deviating navigator keys (NightmareJS browser instance)

MimeTypes

```
application / x - google - chrome - pdf
```

Table C- 12: Deviating MimeTypes (NightmareJS browser instance)

WebSocket

```
close:function close() {
    [native code]
},send:function send() {
    [native code]
},url:ws://localhost:8080/,readyState:0,bufferedAmount:0,onopen:null,onerror:null,onclose
:null,extensions:,protocol:,onmessage:null,binaryType:blob,CONNECTING:0,OPEN:1,CLOSING:2,
CLOSED:3,addEventListener:function addEventListener() {
    [native code]
},removeEventListener:function removeEventListener() {
    [native code]
},dispatchEvent:function dispatchEvent() {
    [native code]
}
```

Code Snippet C- 11: Deviating WebSocket structure (NightmareJS browser instance)

Single browser property deviations

Browser property	Standalone driven browser property	Web-bot driven browser property value
Has lied languages*	False	True
Popup suppression	False	True

Table C- 13: Single browser property deviations (NightmareJS browser instances)

* The property indicates if the language of the browser UI corresponds with sequences of the available browser languages.

Automated Google Chrome / Chromium

User agent

The userAgent string of the headless automated Google Chrome browser instance contains the literal 'HeadlessChrome'.

Canvas

The rendering of the *canvas* DOM element differs in many ways in comparison of the *canvas* DOM element rendered by the Google Chrome browser instance.

Windows keys

window_keys	Missing
Automated Chrome (HL)	chrome attr
Automated Chrome (HF)	attr
Automated Chromium (HL)	chrome

Table C- 14: Deviating window keys (Automated Google Chrome / Chromium browser instance)

Navigator attributes

Navigator keys	Added	Modified
Automated Chrome (HL) Automated Chromium (HL)	webdriver	appVersion userAgent
Automated Chrome (HF) Automated Chromium (HF)	webdriver	

Table C- 15: Deviating navigator attributes (Automated Google Chrome / Chromium browser instance)

MimeTypes

```
application / pdf, application / x - google - chrome - pdf, application / x - ppapi –
widevine - cdm
```

Code Snippet C- 12: Deviating MimeTypes (Headful Chromium browser instance)

```
application / futuresplash, application / pdf, application / x - google - chrome - pdf, a
pplication / x - nacl, application / x - pnacl, application / x - ppapi - widevine - cdm,
 application / x - shockwave - flash
```

Code Snippet C- 13: MimeTypes (standalone Google Chrome / Chromium browser instance)

Single browser property deviations

Browser property	Standalone driven browser property	Web-bot driven browser property value
Resolution	1920x975	800x600
Available resolution	1920x975	800x600
Has lied languages*	False	True
Popup suppression	False	True

Table C- 16: Single browser property deviations (Automated Google Chrome / Chromium browser instance)

* The property indicates if the language of the browser UI corresponds with sequences of the available browser languages.

C.2 Gecko + Spidermonkey

This paragraph described the deviating browser property of web-bot driven browser instances compared against a standalone Mozilla Firefox browser instance.

Single browser property deviations

Browser property	Standalone driven browser property	Web-bot driven browser property value
Resolution	1920x975	1366x768
Available resolution	1920x975	1366x768

Table C- 17: Single browser property deviations (Selenium + Firefox browser instance)

C.3 Trident + JScript

This paragraph described the deviating browser property of web-bot driven browser instances compared against a standalone Internet Explorer browser instance.

User agent
The userAgent property of the Selenium driven browser instance is slightly different compared to the standalone driver browser instance as illustrated in Code Snippet C- 14 and Code Snippet C- 15.
The WOW64 value stands for Windows On Windows which represents an emulator that runs on Windows 64bit operating system[72].

```
Mozilla / 5.0(Windows NT 10.0; WOW64; Trident / 7.0;.NET4.0 C;.NET4.0E;.NET CLR 2.0.50727
;.NET CLR 3.0.30729;.NET CLR 3.5.30729; rv: 11.0) like Gecko
```

Code Snippet C- 14: Deviating userAgent (Selenium + Internet Explorer browser instance)

Internet explorer

```
Mozilla / 5.0(Windows NT 10.0; Win64; x64; Trident / 7.0;.NET4.0 C;.NET4.0E;.NET CLR 2.0.
50727;.NET CLR 3.0.30729;.NET CLR 3.5.30729; rv: 11.0) like Gecko
```

Selenium + Internet explorer

Code Snippet C- 15: userAgent property (standalone Internet Explorer browser instance)

CPU class / Navigator platform
The CPU class is different from the standalone Internet Explorer browser as it indicated x86 instead of x64. Reason for the difference is the 32 build browser webdriver that is being used to improve the speed of the Selenium installation. We encountered some problems with the 64 version in combination with entering text in the input box of our text page. The same argument holds in respect to the difference in the *navigator_platform* value.

Window keys

Missing
__BROWSERTOOLS_CONSOLE_SAFEFUNC
__BROWSERTOOLS_CONSOLE_BREAKMODE_FUNC
__BROWSERTOOLS_CONSOLE
__BROWSERTOOLS_EMULATIONTOOLS_ADDED
__BROWSERTOOLS_DOMEXPLORER_ADDED
__BROWSERTOOLS_MEMORYANALYZER_ADDED
$0
$1
$2
$3
$4
__BROWSERTOOLS_NETWORK_TOOL_ADDED
__BROWSERTOOLS_DEBUGGER

Code Snippet C- 16: Deviating window keys (Selenium + Internet Explorer browser instance)

[72] http://www.thundercloud.net/infoave/new/what-is-wow64/

Document keys

Missing	Added
__IE_DEVTOOLBAR_CONSOLE_EVAL_ERROR __IE_DEVTOOLBAR_CONSOLE_EVAL_ERRORCODE	__webdriver_script_fn

Code Snippet C- 17: Deviating document keys (Selenium + Internet Explorer browser instance)

C.4 EdgeHTML + Chakra

This paragraph described the deviating browser property of web-bot driven browser instances compared against a standalone Edge browser instance.

Single browser property deviations

Browser property	Standalone driven browser property	Web-bot driven browser property value
Popup suppression	False	True

Code Snippet C- 18: Single browser property deviations (Selenium + Edge browser instance)

Navigator attributes

Navigator keys	Missing	Added	Modified
Selenium Edge			webdriver

Code Snippet C- 19: Deviating navigator attributes (Selenium + Edge browser instance)

D. Code contributions

Below are all the code contributions listed relating to the corresponding research phase described in chapter 4.

D.1 Analysis of a client side web-bot detection implementation

Application: Chromium project.
URL: https://chromium.googlesource.com/chromium
Code file:

File	Lines of code (LOC)	Description
/src/chrome/test/chromedriver/js	9	Replaces the static value of the DOM global document object key '$cdc_asdjflasutopfhvcZLmcfl_' with a random generated constant value. Its purpose is to avoid web-bot detection implemented on StubHub.com.

D 1: Modified code file used during the phase: analysis of a client side web-bot detection implementation

D.2 Determining the web-bot fingerprint surface

Application: Web site dedicated to extract browser properties by making use of passive and active fingerprinting.
URL: https://github.com/GabryVlot/BrowserBasedBotFP
Code files:

Files	Lines of code (LOC)	Description
/settings.json	1	Contains the root path of the application used for folder reference for persisting the browser properties
/tests/diffFP_test.js	10	Contains a unit test to test the delta comparison function between the properties of two different browsers.
/test/test_data.js	659	Contains the test data for the unit test in /tests.diffFP_test.js
/server/fingerprint.js	129	Responsible extracting FP properties from HTTP Body originating from Client and persisting them into DB.
/server/server.js	61	Contains Express Webserver configuration and initialisation including routes.
/public/css/index.css	2	CSS markup for the homepage of the web site used to extract the browser properties.
/public/js/fingerprint.js	141	Extracts browser fingerprint from Global DOM objects.
/public/js/utils.s	81	Contains general functions to aid the abstraction of the browser fingerprint.
/public/swf/swfobject.js	6	SWF object to include on the website to test Flash specific browser properties.
/public/FontList.swf	*	SWF object to include on the website to test Flash specific font browser properties

/db/db-utils.js	156	Responsible for the creation of the DB structure and making the DB connection
client/index.html	65	Home page.
analyse/diffFP.js	220	Compares FP results based upon their IDs and outputs the differences.

D 2: Modified code files used during the phase: Determining the web-bot fingerprint surface

D.3 Measuring the adoption of web-bot detection on the internet

Application: Web-bot detection scanner. Measures the adoption of web-bot detection on the internet by making use of detection patterns based upon a web-bot fingerprint surface.

URL: https://github.com/GabryVlot/BrowserBasedBotFP
Code files:

Files	Lines of code (LOC)	Description
/scan.py	65	Initiating browsers and browser commands. Responsible for looping over the Alexa 1 Million by making use of multiple browsers running in parallel.
/scanLocal.py	55	File dedicated for 'offline' testing of JavaScript file without using a browser but still using the full web-bot detection mechanism.
/detection/Script.py	63	Entity: Represents a web page script
/detection/Scanner.py	124	Extracts inline and external script code from page content and analyses the data for bot detection inclusions
/detection/RegisteredDetectionTopic.py	17	Entity: Represents a detection topic e.g. BotDetectionProperties_selenium that aggregates the found patterns in this category and also the total detection score.
/detection/PatternChecker.py	78	Analysis script content by making use of regular expressions.
/detection/FileManager.py	171	Responsible for all file operations (downloading, de-obfuscating and writing to disk).
/detection/DetectionPatternFactory.py	15	Responsible for the creation of a DetectionPattern instance.
/detection/DetectionPattern.py	16	Embodies a detection pattern that will be used to identify web-bot detection scripts / sites
/detection/BotDetectionValueMangaer.py	39	Is responsible for calculating the bot detection value
/detection/validation/*.csv	*	Contains 25 files with comma separated values that can be used for validation of the web-bot detection scanner. All csv files contain web site URLs of cyber security companies that show signs of web-bot detection.
/detecting/tests/decompress.csv	*	File containing comma separated values embodied in URL of web sites for

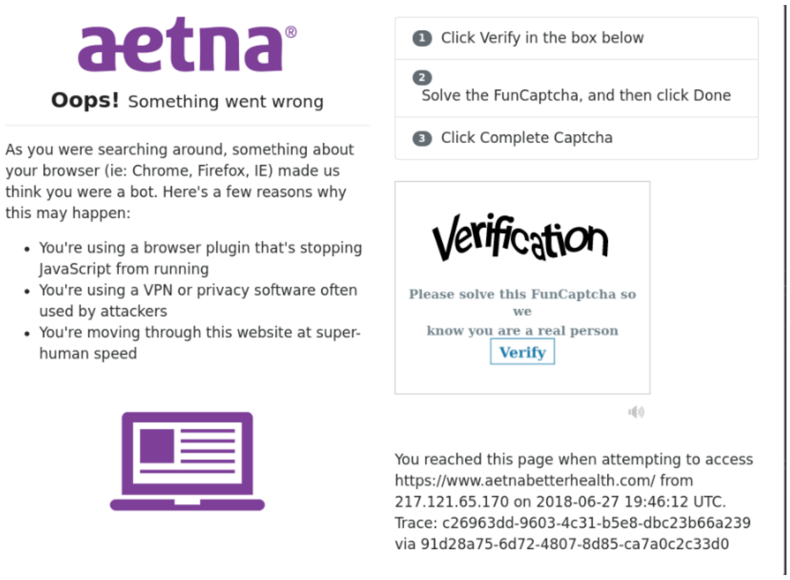
		testing decompression of HTTP body content.
/detection/examples	*	Contains 25 folders of identified web-bot detection implementations per library / detector.
/detection/detectionPatterns/ ManuallyFoundLiterarls.py	67	Contains repeating detection patterns across different sites.
/detection/detectionPatterns/ DetectorPatterns.py	29	Contains detection patterns based on found web-bot detection script inclusions.
/detection/detectionPatterns/ BrowserFingerprints.py	65	Contains detection patterns based on web-bot specific browser properties (Characteristics).
/detection/detectionPatterns/ BotFingerprintingSurface.py	49	Contains detection patterns based on web-bot specific browser properties.
/detection/detectionPatterns/db/DB.py	98	Responsible for persisting scripts and detection patterns.
/detection/configuration/config.json	7	Contains the configuration of excludeFiles like for instance jquery.
/detection/alexa/top-1m.csv	*	Contains the top 1 million web sites that we used for measuring the adoption of web-bot detection on the internet.

D 3: Measuring the adoption of web-bot detection on the internet

E. Observed web page deviations

In this appendix we give an overview of observed deviations per deviation type when visiting web-bot detection sites with a web-bot browser instance (cf. chapter 9).
If applicable; the left side of the illustrations represent the page visit by a standalone browser and the right side represents the web page visit by a web-bot browser instance.

E.1 Blocked content



aetna®

Oops! Something went wrong

As you were searching around, something about your browser (ie: Chrome, Firefox, IE) made us think you were a bot. Here's a few reasons why this may happen:

- You're using a browser plugin that's stopping JavaScript from running
- You're using a VPN or privacy software often used by attackers
- You're moving through this website at super-human speed

1 Click Verify in the box below

2 Solve the FunCaptcha, and then click Done

3 Click Complete Captcha

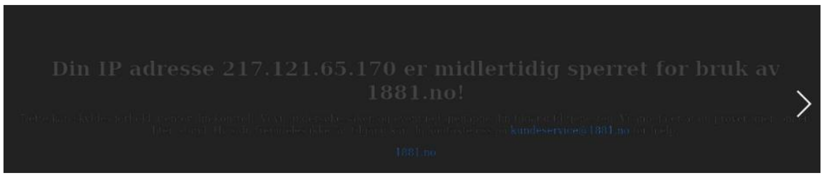
Verification

Please solve this FunCaptcha so we know you are a real person

Verify

You reached this page when attempting to access https://www.aetnabetterhealth.com/ from 217.121.65.170 on 2018-06-27 19:46:12 UTC.
Trace: c26963dd-9603-4c31-b5e8-dbc23b66a239 via 91d28a75-6d72-4807-8d85-ca7a0c2c33d0

E 1: Blocked content by Captcha on *http://diabetesincontrol.com*



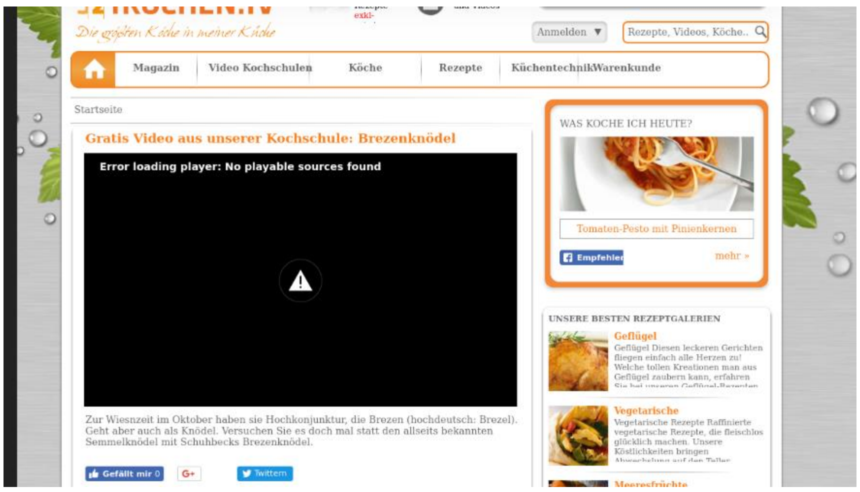
Din IP adresse 217.121.65.170 er midlertidig sperret for bruk av 1881.no!

[illegible] kundeservice@1881.no [illegible]

1881.no

E 2: Blocked content by message on http://eiendomspriser.no

E.2 Deviation in the content of the web page.



Die größten Köche in meiner Küche

Anmelden ▼ | Rezepte, Videos, Köche..

Magazin | Video Kochschulen | Köche | Rezepte | KüchentechnikWarenkunde

Startseite

Gratis Video aus unserer Kochschule: Brezenknödel

Error loading player: No playable sources found

Zur Wiesnzeit im Oktober haben sie Hochkonjunktur, die Brezen (hochdeutsch: Brezel). Geht aber auch als Knödel. Versuchen Sie es doch mal statt den allseits bekannten Semmelknödel mit Schuhbecks Brezenknödel.

Gefällt mir 0 | G+ | Twittern

WAS KOCHE ICH HEUTE?

Tomaten-Pesto mit Pinienkernen

Empfehlen | mehr »

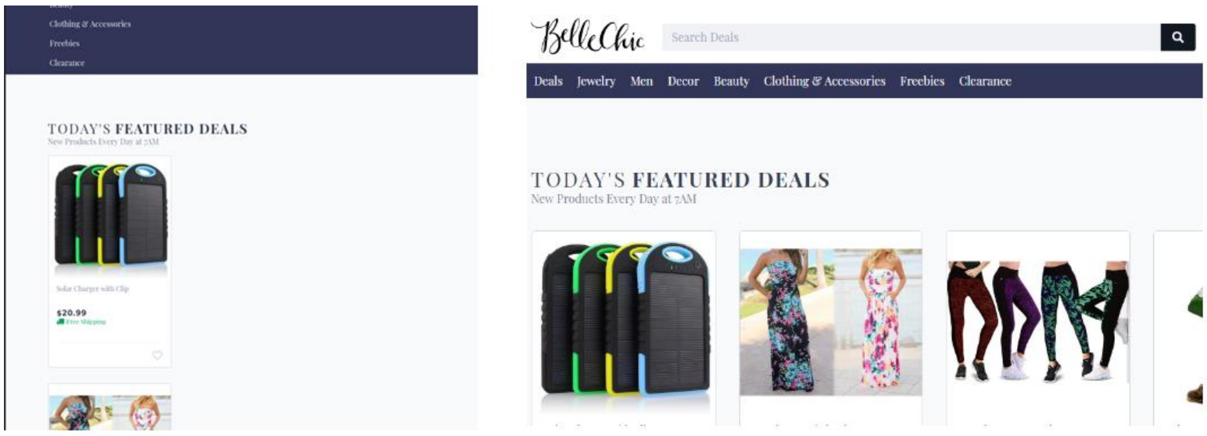
UNSERE BESTEN REZEPTGALERIEN

Geflügel
Geflügel Diesen leckeren Gerichten fliegen einfach alle Herzen zu! Welche tollen Kreationen man aus Geflügel zaubern kann, erfahren

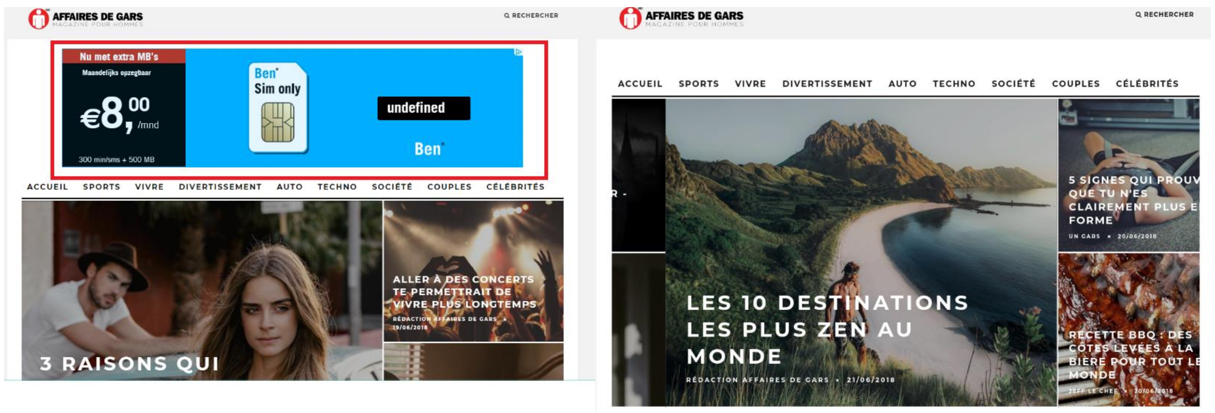
Vegetarische
Vegetarische Rezepte Raffinierte vegetarische Rezepte, die fleischlos glücklich machen. Unsere Köstlichkeiten bringen

Meeresfrüchte

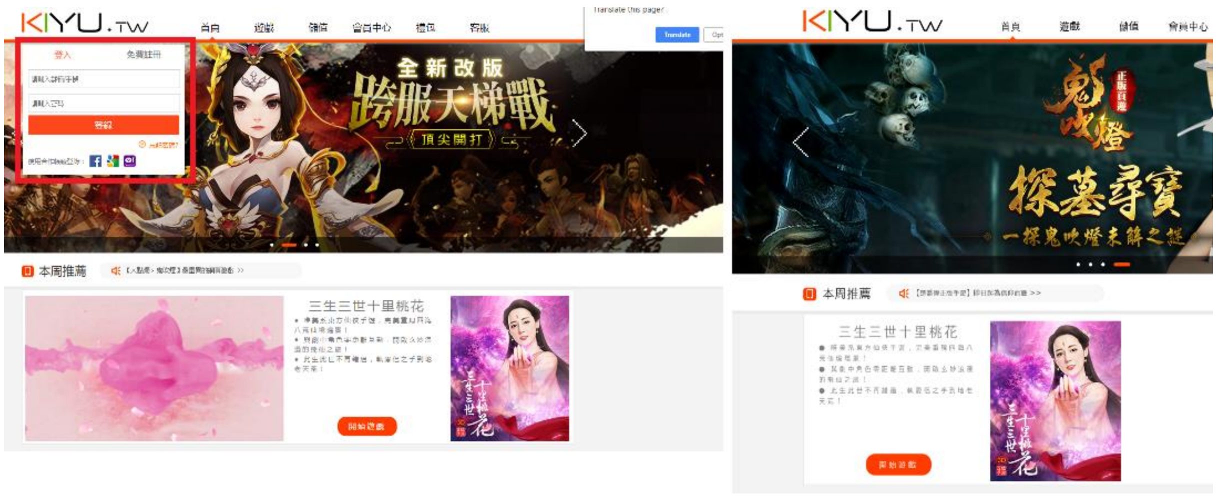
E 3: Video cannot be loaded on http://321kochen.tv



E 4: Deviation in web page layout on http://bellechic.com


AFFAIRES DE GARS
RECHERCHER
Nu met extra MB's
€8,00 /mnd
Ben Sim only
undefined
Ben
ACCUEIL
SPORTS
VIVRE
DIVERTISSEMENT
AUTO
TECHNO
SOCIÉTÉ
COUPLES
CÉLÉBRITÉS
3 RAISONS QUI
ALLER À DES CONCERTS TE PERMETTRAIT DE VIVRE PLUS LONGTEMPS
LES 10 DESTINATIONS LES PLUS ZEN AU MONDE
RÉDACTION AFFAIRES DE GARS • 21/06/2018
5 SIGNES QUI PROUV QUE TU N'ES CLAIREMENT PLUS E FORME
RECETTE BBQ : DES CÔTES LEVÉES À LA BIÈRE POUR TOUT LE MONDE

E 5: Advertisements are not loaded on http://affairesdegars.com



E 6: Login dialog is not visible on http://kiyu.tw

Bibliography

Scientific references

[AJN+13] Acar G, Juarez M, Nikiforakis N, Diaz C, Gürses S, Piessens F, et al., editors. FPDetective: dusting the web for fingerprinters 2013: ACM.

[AMS+10] Ager B, Mühlbauer W, Smaragdakis G, Uhlig S, editors. Comparing DNS resolvers in the wild2010: ACM.

[CMF+11] Catanese S, De Meo P, Ferrara E, Fiumara G, Provetti A, editors. Crawling Facebook for social network analysis purposes 2011: ACM.

[DNV15] Dai K, Nespereira CG, Vilas AF. Scraping and Clustering Techniques for the Characterization of Linkedin Profiles. Computer Science & Information Technology. 2015;5(1):1-15.

[ECK10] Eckersley P, editor How Unique Is Your Web Browser?2010; Berlin, Heidelberg: Springer Berlin Heidelberg.

[EN16] Englehardt S, Narayanan A. Online Tracking: A 1-million-site Measurement and Analysis. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security; Vienna, Austria. 2978313: ACM; 2016. p. 1388-401.

[HHC10] Hsu C-H, Huang C-Y, Chen K-T. Fast-Flux Bot Detection in Real Time. 6307. Berlin, Heidelberg: Springer Berlin Heidelberg; 2010. p. 464-83.

[LCY+08] Liu L, Chen S, Yan G, Zhang Z. BotTracer: Execution-Based Bot-Like Malware Detection. 5222. Berlin, Heidelberg: Springer Berlin Heidelberg; 2008. p. 97-113.

[MNM17] Mudialba PJ, Nair S, Ma J. Finger printing on the web. UEMCON; 19-21 Oct. 2017; New York City, NY, USA. conf/UEMCON/MNM17: IEEE; 2017. p. 475-6.

[NKJ+13] Nikiforakis N, Kapravelos A, Joosen W, Kruegel C, Piessens F, Vigna G. Cookieless Monster: Exploring the Ecosystem of Web-Based Device Fingerprinting. Proceedings of the 2013 IEEE Symposium on Security and Privacy. 2498133: IEEE Computer Society; 2013. p. 541-55.

[RBB+10] Rydstedt G, Bursztein E, Boneh D, Jackson C. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In IEEE Oakland Web 2.0 Security and Privacy (W2SP '10), Oakland, CA, May 2010.

[RKD10] Rieck K, Krueger T, Dewald A, editors. Cujo: efficient detection and prevention of drive-by-download attacks 2010: ACM.

[SIM17] Simonite, T. (2017). Study Suggests Google's Ad-Targeting System May Discriminate. [online] MIT Technology Review. Available at: https://www.technologyreview.com/s/539021/ probing-the-dark-side-of-googles-ad-targeting-system/ [Accessed 15 Aug. 2017].

[TJM14] Torres CF, Jonker HL, Mauw S. FP-Block: Usable Web Privacy by Controlling Browser Fingerprinting. In: Pernul G, Ryan PYA, Weippl ER, editors. ESORICS (2). conf/esorics/TorresJM15: Springer; 2015. p. 3-19.

[TK02] Tan P-N, Kumar V. Discovery of Web Robot Sessions Based on their Navigational Patterns. Data Mining and Knowledge Discovery. 2002;6(1):9-35.

[YXK10] Yu F, Xie Y, Ke Q, editors. SBotMiner: large scale search bot detection2010: ACM.

[YZW+12] Yajin Z, Zhi W, Wu Z, Xuxian J, editors. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative {Android} Markets. Proceedings of the 19th Annual Network Distributed System Security Symposium; 2012 Februari 2012. San Diego, CA, USA.

Books

[Sch12] Schrenk M. Webbots, spiders, and screen scrapers; a guide to developing internet agents with PHP/CURL, 2d ed: 1. Portland: Ringgold Inc; 2012.

Commercial publications

[Whi16] WhiteOps. The Methbot Operation (white paper). 2016. p. 30. Available from: http://w-ops.com/methbot_wp.

[Dis16] Distil Networks. The 2016 Bad Bot Landscape Report (white paper). 2016. p.16. Available from: https://resources.distilnetworks.com/white-paper-reports/2016-bad-bot-report.

Glossary

Browser characteristics Characteristics of the browser like for instance installed plugins.

Browser fingerprint Browser property that can be used to uniquely identify a browser instance.

Browser family fingerprint Combination of used layout and JavaScript engine that uniquely can identify a browser family.

Browser properties All properties exposed by a browser instance through the global DOM objects *window, navigator* and document.

ChromeDriver A standalone server which implements WebDriver's wire protocol for chromium.

Compare configurations Configurations to compare the browser properties of a web-bot driven and standalone browser instance.

De-obfuscation Parsing of obfuscated String to UTF-8 readable format.

Detection pattern A detection pattern contains a pattern that is being used to calculate the detection score and find web-bot detection script inclusions

Deviation An observable difference on a web page caused by the employment of a web-bot in comparison to the web page visited by a human being.

Global DOM object Global *window*, *navigator* or *document* object exposed by the DOM API.

Standalone browser Browser driven by a human being.

Server host Owner of a web site or the owner of a library included on the web site.

Visitor class Class used to indicate a visitor: web-bot or human.

Web-bot Automated agent that uses specific software to extract data from web pages.

Web-bot based studies Studies that employ a web-bot to automatically extract data from web pages.

Web-bot detection Detection of a web-bot by a web site.

Web-bot detection level Indicates the level of web-bot detection per script inclusion.

Web-bot fingerprinting surface Specific browser properties that can be used to detect a web bot.

Web-bot detection sites Web sites that contain scripts with web-bot detection code inclusions.

Web-bot detector Stakeholder that is interested in the detection of a web-bot and is able to detect a web bot.

Web-bot fingerprint Browser properties that are strongly relating and only exist when a browser instance is driven by a web bot.

WebDriver An open source tool for automated testing of webapps across many browsers.

IMAGE CREDITS: COLIN ANDERSON / GETTY IMAGES