

École polytechnique de Louvain

Online bot detection using real-time Machine Learning classification.

Author: **Robin-Gilles BECKER**
Supervisors: **Ramin SADRE, Tom BARBETTE**
Reader: **Cristel PELSSER**
Academic year 2025–2026
Master [120] in Cybersecurity

I hereby confirm that this thesis was written independently by myself without the use of any sources beyond those cited, and all passages and ideas taken from other sources are cited accordingly. Generative AI was not used in the writing of this thesis. Generative AI was used for text generation in the making of the two honeypot sites.

The author(s) gives (give) permission to make this master dissertation available for consultation and to copy parts of this master dissertation for personal use. In all cases of other use, the copyright terms have to be respected, in particular with regard to the obligation to state explicitly the source when quoting results from this master dissertation.

23/03/2026

Abstract

The dead internet theory states that the majority of internet traffic is generated by bots instead of humans. With the advent of AI, this theory has become reality. Myriads of bots are sent accross the web to crawl and scrape to feed AI datasets. This has become a massive problem for website owners who see their content stolen and server load increase drastically. Bot detection and protection has been a topic of research as long as bot traffic has existed but no single miracle solution has yet been found. From Honeypots and Captchas to Fingerprinting and obfuscation, each known method has advantages and disadvantages when it comes to their efficiency, impact on normal users and uses in specific use-cases.

In this paper, we propose a single-request on-the-fly bot detection method using machine learning with a K-Nearest Neighbors model and a Random Forest model trained on HTTP requests from Honeypots and logs from the University of Louvain-la-Neuve. Our objective for the machine learning is to outperform rule-based methods with the goal of detecting bots to reduce server load on the University's network.

To that end, we start by reviewing the research that has been performed until now in order to gauge the efficiency of existing methods and understand what goes into designing one. After that, we build Honeypots to catch bot traffic, perform an analysis on the logs to design a rule-based benchmark and finally develop and test multiple machine learning detection system options.

In the end, we are able to filter bots from humans from a single HTTP request with better accuracy than rule-based systems but holes in the training data still has an impact on our results.

Keywords: Webcrawlers, Bots, Machine learning, Bot detection, Honeypot, Internet traffic

Acknowledgements

I would like to thank my thesis supervisors who guided me during my last year of Master in Cybersecurity: Prof. Ramin Sadre and Prof. Tom Barbette. I also would like to thank Cristel Pelsser, reader and member of the jury, and Rémi Van Boxem.

More personally, I would like to thank my friends and family for the support.

Table of Contents

Abstracts	I
Abstract	I
Table of Contents	IV
List of Figures	V
List of Tables	VI
1 Introduction	1
1.1 Motivations	1
1.1.1 The consequences	1
1.1.2 Technological context	2
1.2 Problem statement	3
1.2.1 Why is this problem hard to solve?	3
1.2.2 Our contribution	3
1.3 Organization of this document	4
2 Literature review, definitions, state of the art and related work	5
2.1 Analysis of webcrawler behavior	5
2.1.1 The "What"	5
2.1.2 The "Who" and the "Why"	9
2.1.3 The How?	10
2.2 Analysis of protection mechanisms	11
2.2.1 Content Walling	11
2.2.2 Protocol Standards	11
2.2.3 Reputation Lists	14
2.2.4 Honeypots	15
2.2.5 CAPTCHAS & Challenges	18
2.2.6 Content Obfuscation	19
2.2.7 Fingerprinting	22
2.2.8 Machine Learning	23
2.2.9 Content Delivery Networks	26
2.3 After the detection	27
2.3.1 Bans and limitations	27
2.3.2 Aggressive Methods	27
2.4 Review summary	28
3 Methodology & Implementation	30
3.1 Design Requirements	30
3.1.1 Benchmark	31
3.2 Honeypot deployment	31

3.2.1	Honeysite design	31
3.2.2	Honeysite creation	32
3.2.3	Setting up tests	33
3.2.4	Deployment & Monitoring	36
3.3	Traffic log analysis	36
3.3.1	General statistics	36
3.3.2	Deeper analysis	37
3.4	Feature Extraction	41
3.4.1	URI and Status Code	42
3.4.2	Sec Fields	43
3.4.3	User Agent	44
3.4.4	Summary of features	46
3.5	Implementation of the Machine Learning Model	46
3.5.1	Benchmark	46
3.6	Architecture	47
3.6.1	Semi-Supervised Learning	48
3.6.2	Supervised Learning	54
3.6.3	Final Improvements	58
4	Results	63
4.1	Benchmark	63
4.2	Accuracy	63
4.2.1	More realistic set	64
4.2.2	Server-agnostic model	67
4.3	Time	68
5	Discussion	71
5.1	False positive on diverse human dataset	71
5.2	Comparison with state of the art/related works	71
5.2.1	Why those results?	72
5.3	Limitations	74
5.3.1	Efficiency at scale	74
5.3.2	Evolutions of technology	75
5.3.3	Present usability	75
5.4	CyberSecurity analysis of the project	76
6	Future work	78
7	Conclusion	79
	Bibliography	85
	Appendices	86

List of Figures

2.1	Architecture of a webcrawler ¹	6
2.2	Architecture of a crawl frontier ²	6
2.3	Tests of companies' claims about their respect of robots.txt.	10
3.1	Architecture of the Machine Learning process	47
3.2	First results with KNN	55
3.3	First results with RF	56
3.4	First results with LogReg	56
3.5	First results with OCSVM	58
3.6	First results with SVC	58
3.7	Results with MaxAbsScaler	60
3.8	Results with Standard Scaler	61
4.1	Error rates on 50 tests with KNN	64
4.2	Error rates on 50 tests with RF	65
4.3	Error rates on 50 tests: summary	66
4.4	50 tests of the time (seconds) taken to make 3400 classifications with KNN	69
4.5	50 tests of the time (seconds) taken to make 3400 classifications with RF	70

List of Tables

2.1	The who and why.	9
2.2	List of Reputation List services used by researchers in this section . .	15
2.3	Ways to hide elements using CSS	17
2.4	Summary of defenses against bots	29
3.1	Types of bots in the logs of Keta	37
3.2	First results with PUL	52
3.3	PUL results after Grid Search	53
3.4	Impact of features in error rates. Percentage ranges are used in case of significant variations.	59
4.1	Results with hardcoded rules	63
4.2	Average results for both methods	65
4.3	False Positive rates for test set with diverse human data accross 50 tests	66

¹Image by User:ChaTo based on image from PhD. Thesis of Carlos Castillo, image released to public domain by the original author.

²By Lequintanilla - Own work, CC BY-SA 4.0

4.4	Comparison of the models with all the features against the ones adapted to minimize the side effects of poor data diversity.	68
4.5	Feature comparison of the models	68
4.6	Tests without features enabled by Caddy-specific headers.	68
4.7	Time statistics for both models	69
5.1	Comparison of different methods.	72

Chapter 1

Introduction

1.1 Motivations

Filtering bot from human traffic on the internet has been a long running issue ever since the first crawlers were created. But with the recent development of generative Artificial Intelligence (AI), AI companies have unleashed swarms of bots all across the internet to indiscriminately scrape content from each and every websites with the objective of feeding the datasets used to train their AIs.

Big companies are not the only bot developer profile. In the age of the AI goldrush, everyone tries to build their own, scraping websites to do so. Online artists are also victims of such scraping, as their art is stolen to train image generation AIs, sometimes for large companies, sometimes by rivals launching targeted attacks or potential clients preferring training an AI on a specific artist's work rather than buying art from them. Many different sectors are victims of malicious bots. The bot traffic question - knowing which click is from a bot or a human - has been one of the biggest problem for the online marketing sector for as long as bots existed and it is only getting worse. Webscraping is not only about bots stumbling on a website and downloading its content, it can also consist of targeted attacks: bot developers sending their bots on specific sites to get their hands on specific data. An old but still relevant example of that are price scrapers: bots deployed to monitor the prices of items in an online shop to notify their users when prices go down. But with generative AI (genAI) easily available to anyone, all websites can become victims of targeted attacks. Rival websites can be generated fast and the content of the target website can be scraped, reworded by genAI and reposted to the rival website using a completely automated process resulting in the rapid propagation of what has been named "AI slop".

1.1.1 The consequences

Online bot activity is more than just annoying, it is deleterious in itself due to its many negative consequences. We touched on the obvious copyright infringement issues but more critical is the environmental impact. As a 2024 Nature article estimated [59], 1.8% to 2.8% of the global pollution was due to digital technologies, including internet traffic. This is more than the global anthropogenic greenhouse emissions of most countries. And it goes higher according to other studies. Meanwhile, recent statistics from Thales [38] evaluated the proportion of bot activity to exceed 51% of the total internet traffic. The dead internet theory, which states that the majority of web traffic is bot-made, has thus finally become true. Thales highlights the role of AI and Large Language Models (LLMs) in this increase, as they have simplified the creation and scaling of bots for malicious purposes. Not all bots are malicious but

from 2023 to 2025, they noted a jump from 32% to 37% of malicious bots' share of all internet traffic. When the global anthropogenic greenhouse emissions numbers from 2024 were released, humans were still the majority of online traffic. Now that the proportion of bots has exceeded humans and does not show any sign of slowing down, how will that affect the environmental footprint of the digital sector? And this is only accounting for traffic. Data storage too, has an impact. Tech companies need to accelerate the construction of datacenters, each of them being an energy blackhole, to store both their AIs and the slop they unceasingly produce.

Even not looking at the big picture and only zooming on the websites themselves, bot traffic drastically increases server load, which in turn increases operational costs. The result is an equivalent of a slow-burn DDOS attack or the shutdown of the website for economic reasons. This is especially true since Thales's data indicates the amount of simpler bot attacks has recently surpassed the amount of advanced targeted bot attacks in some sectors like the travel industry (52% vs 41%) due to AI tools lowering the barrier of entry for launching bots.

If it was possible to impede webcrawler bots at large scale, website owners could not only protect their content, but the practice of scraping itself might stop being worth the cost and the simpler bot attacks could be rendered ineffective.

1.1.2 Technological context

Online bots and the protection against them are far from being recent topics of research, although their recent proliferation has made the general population more conscious of the issue. Many defenses against bots have been proposed over the years, with different degrees of effectiveness. But the field continues to be a cat-and-mouse game where new defenses are designed to stop bots that keep on bypassing previous defenses.

In the absolute, bots are just a tool to process publicly available online data faster than a human could legitimately read it all. The bot landscape is very varied with many research teams with no malicious intentions trying to improve bot performances. After all, webscrapers are often used for collecting data for research purposes. However, a large portion of bot traffic also includes vulnerability scanners who crawl the web looking in websites for vulnerabilities to exploit for malicious purposes. On the other hand, many bots, such as indexing bots, are benign and their presence is even desired.

An issue we observe with bot detection is that the constantly evolving technologies make it so maintaining a detection system requires much work from the webmasters. Indeed, many methods are based on the current state of bot technology, standards, or the state of many other components such as the OS and browser. For this reason and because no one single solution exists, each web developer ends up developing their own methods, with none being nowhere close to being considered as "widely used".

1.2 Problem statement

Because of the bot traffic on the network of the University of Louvain-La-Neuve (UCL), server load has increased significantly. We want to design a bot detection method that would be able to detect bot traffic with high accuracy in order to reduce their impact. We also want the proposed method to be able to work on other websites for general bot protection purposes. As we will talk about, targeted attacks require defenses specific to the targeted website that a general purpose defense system cannot address without bloating itself.

1.2.1 Why is this problem hard to solve?

Ideally, the perfect detection system would filter out bots as soon as possible without affecting legitimate human browsing. This implies no deterioration of the browsing experience and no false positives. In addition, the ideal method would be able to adapt alongside adverse technologies to catch bots with evolving behavior. And that without requiring substantial amount of work from the website owner, neither costing significant resources.

Due to the high impact of bots in the sector of online marketing and cybersecurity, among others, a lot of research on the protection against bots has been conducted. Despite this fact, this issue is still unsolved. Reasons are plenty. The most advanced the bot, the better it is at camouflaging as a real human. It can spoof its attributes, dynamically change its behavior, notice when it has trigger defense systems and more. What makes it worse when it comes to targeted attacks is that the bigger the impact they have on the website, the more likely the bots carrying them are to have advanced features to bypass defenses. In this arms race, the threat landscape and attacker capabilities are everchanging and protection methods have a hard time staying up to date. General defenses are easier to update but will only catch basic bots while defenses specific to a use-case or a type of attack have to constantly stay ahead to be effective.

One method able to learn and adapt to new behavior is Machine Learning. However, the difficulty resides here in the gathering of data to feed the training datasets. Collection is hard, as guaranteeing that no bot is present in the human data (and conversely) is non-trivial because it essentially demands to solve the detection problem in the first place or to find a collection method that can guarantee data purity. As we will discover later in our research, no public dataset with relevant data currently exist.

While no perfect solution currently exist, we believe that several defenses working together might fill each other's gaps.

1.2.2 Our contribution

Although crafting the perfect defense method as described in the previous subsection is still unrealistic at the moment, we can try to get as close to this objective as

possible.

To do so, we study the state of the art to see how each existing defense technique performs on those metrics and use that knowledge to make our own. Our proposed method contributes to the research by classifying HTTP requests using data points that have not been used before and by attempting to perform this classification based on a single request while such classification has generally been done on whole sessions at the time of writing. In particular, we make use of the Cloudflare Worker header fields, Chrome’s Sec-CH-UA related fields and design a new way to extract features from the User Agent to be used by a K-Nearest Neighbors model. For our training and testing data, we also decided to focus on accuracy by choosing data that we can be certain of their label (bot or human) instead of relying on close-enough heuristics. While this reduces the amount of available data, we gain in accuracy, as bots have become so good at mimicking human traffic that labelling data based on heuristics and rules is no longer reliable.

1.3 Organization of this document

In this paper, we will start by analysing the set of existing detection and protection methods against web bots present in the state of the art. We will also learn about how webcrawlers work and the strategies they use, so that we can gauge the effectiveness of the existing methods against them and later address them accordingly in our own. After comparing the pros and cons of the many methods, we will see how each different use-cases has different requirements, why it is important to take that into account and how they will impact the design of our own detection method. From this, we will propose our detection method and start building it. This will include the gathering of data using the honeysites that we will create, the analysis of this data to inform and confirm the data from the state of the art and our hypothesis, and the coding of the method. After tuning our solution until it presents satisfying results, we will discuss those results and address the potential gaps.

Chapter 2

Literature review, definitions, state of the art and related work

In this chapter, we will review research papers talking about the webcrawler issue. We will learn about how those bots work, what protection measures have already been attempted and how bots attempt to evade them in return. At the end of this chapter, we will have a good view of the webcrawler defense landscape, what works and what does not, and we will be ready to conceptualize our own defense to enter this cat and mouse game.

The war on webcrawler being currently ongoing, the research moves fast and new technology is continuously being developed. For this reason, it is important to situate our work in time and note that this state of the art analysis has been made at the start of this project, in September 2026.

2.1 Analysis of webcrawler behavior

In order to design effective protections against webcrawlers, we first need to understand what they are made of, who creates them and how they act. In this section, we will ask the what, who, why and how of webcrawler bots.

2.1.1 The "What"



Definitions

Webcrawlers (sometimes called a spider, spiderbot, ant or web scutter): automated programs who navigate through websites, and visit all links of interest [75]. Architecture represented in Figure 2.1.

Webscraper: software that automatically gathers desired information from a website, normalizing it into a format that is useful for the intended purpose, and storing in an offline database for later use [60]. They are often built on top of a crawler to allow data gathering from multiple pages on the website.

Seeds: initial URLs given to the webcrawler for it to visit.

Crawl Frontier: data structure, often a queue, used for storage of URLs eligible for crawling, and supporting such operations as adding/removing URLs and selecting which ones to crawl according to custom policies [45]. See Figure 2.2.

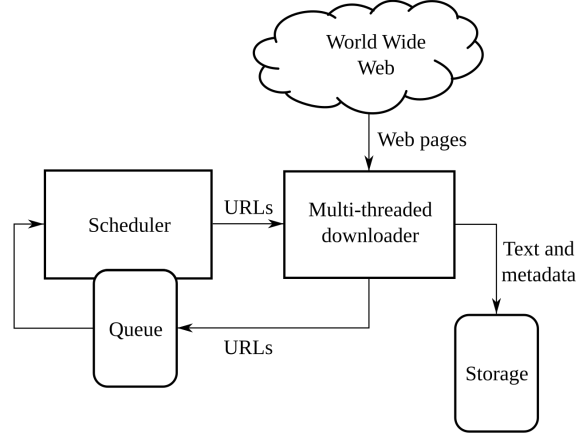


Figure 2.1: Architecture of a webcrawler¹

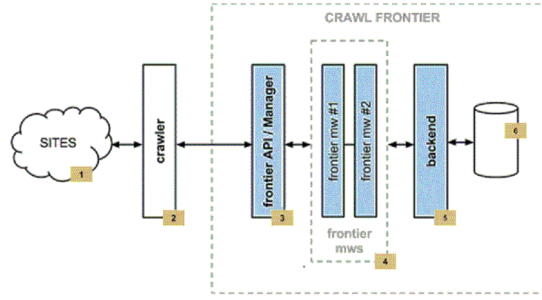


Figure 2.2: Architecture of a crawl frontier²

The typical workflow of a crawler [75] is as follow:

1. Setup: to begin crawling, crawlers are given a starter list of seeds.
2. Access: create a connection to the site.
3. Navigation: reach pages of interest by following links or loading more content (e.g. through scrolling).
4. Loading: load the contents of the page.
5. Data gathering: search for further links to follow and/or fetch the specific desired data.

Web pages are visited and hyperlinks present on the pages are added to the crawl frontier. If the hyperlinks are ordered in the frontier according to specific rules, it is called Focused Crawling. Steps 2 to 5 are then repeated, which results in the recursive visit of the web pages contained in the frontier.

At every step of this process, the web admins are able to put protection mechanisms to detect, limit or block the bot, which we will see in the next section. A bot developer must then be able to design its crawler in a way that will bypass the security measures.



Definitions

Focused Crawling: crawling strategy which consists of looking at the Web page context and link structure of a crawled link before assigning it a priority based on the likelihood that the Web page will lead to the topic of interest [66].

Distributed Crawling: crawling strategy where a first scout bot comes to identify links of interest and then distribute the crawling task to a set of worker bots [7]. This allows to use a wider set of IP addresses as a countermeasure to IP blocking. Furthermore, each bot making only a few requests makes it more difficult for the website to identify the crawling operation taking place.

Bot Farm: network of bots, often spread across multiple locations, used to mimic human online interactions.

Crawlers are not always alone, especially in case of a massive crawling operation where bot farms can be used. Sometimes, a horde of bots are simply sent at once to the website. But other times, the strategy is more complex, like with Distributed Crawling.

Selectors

Inside the webscraper family, we note two common types of Selectors [54] [29], which define how the bot looks for data within the pages:

The **XPath Selector**, short for XML Path, is designed to navigate XML documents and find the desired XML and HTML elements within them. XML being one of two major languages used by browsers to display websites. The main appeal of XPath is its ability to navigate the Document Object Model (DOM), the API for HTML and XML documents that defines the logical structure of documents and the way a document is accessed and manipulated. However, XPath-based scrapers are rather complex to program and require a lot of work and testing to make sure they select the correct data. For example, to extract data from a page with this markup:

Listing 2.1: Example of HTML page.

```
<html>
<body>
<h1>Unimportant Title</h1>
<table>
  <tr><td class="details">Data of interest</td></tr>
  <tr><td class="details">Data of interest</td></tr>
  <tr><td class="details">Data of interest</td></tr>
</table>
</body>
</html>
```

The bot would use the following code:

Listing 2.2: Example of XPath Selector code

```
$( '/html/body/table/tr/td' ).text ();
```

The **CSSSelect** takes advantage of the Cascading Style Sheets (CSS) used to make websites prettier. In CSS, HTML elements can be given classes, attributes or IDs and these are what the CSS-based scrapers look for. Programming one is easier and faster than a XPath one, as the developer only needs to specify the CSS elements they are looking for. They are also faster at crawling the page. However, what they gain in speed, they lose in flexibility as they can only go down the DOM, not back up. Differences in navigation style are also the reason why they are more likely to fall into Honeypots placed by the website owners. Using the previous example, a CSS Selector would be coded like this to get the same data.

Listing 2.3: Example of CSSSelector code

```
$( '.details' ).text ();
```

The popular Scrapy framework [25] provides both types of Selectors. Other tools make the practice way more accessible to non-coders by making use of a Graphical User Interface (GUI). Examples are Web content Extractor [32], Mozenda [13] and Easy Web Extract [3].

To make it harder for bots, website developers can periodically change the HTML code of the page by moving elements around or modifying the hierarchy. HTML makes it easy to do so without any apparent visual change to the page. The same goes for changing the CSS, although CSS Selectors are way easier to update so it will not have much impact. Listing 2.4 is a version of the previous example where the source code has been changed to confuse the previous scrapers while retaining the same visual appearance.

Such practices are common for websites with a lot of resources and developers - especially E-commerce sites who want to avoid Price Scrapers - but way too time consuming for smaller websites with less resources.

Listing 2.4: Example of obfuscated HTML page.

```
<html>
<body>
<h1 class="details">Unimportant Title</h1>
<div>
  <table>
    <tr><td class="data">Data of interest</td></tr>
    <tr><td class="data">Data of interest</td></tr>
    <tr><td class="data">Data of interest</td></tr>
  </table>
</div>
</body>
</html>
```

Actor	Goal
Search Engines	Indexing websites to map the internet
AI Companies	Feed their datasets with website data
AI Agents	Answer user question by reading website pages
Application	Provide service to user using website data (e.g. price trackers)
Rival websites	Scraping-pasting the content to their own site
Individuals	Gathering data for their own use

Table 2.1: The who and why.

2.1.2 The "Who" and the "Why"

Anyone with the right tools can create a webcrawler or webscraper for any purpose. If creating one is beyond one's skillset, using a third-party bot service is also possible.

The motivation of the bot developers and users behind their crawling might seem unimportant at first but it will greatly help in the conceptualization of our threat model. Furthermore, the desired goal influence the design of their bots and thus, which defense will be most effective against them. We hereby provide an non-exhaustive list of the goals and kinds of bot users, as described in the literature [63] [29].

The positive about crawlers developed by big companies is that most of them have some sort of documentation with information about their behavior and policies (like Google's [22]), to varying degrees of transparency. In particular, we can take a look at their IP addresses and their claim of respecting the bot policies of the websites they visit, described in their robots.txt. Robots.txt, as we will see in more details in the next section, is a file describing to incoming robots what they can and cannot do on the website. Most companies claim to respect robots.txt. In [63], researchers challenge those claims and present the results in Figure 2.3.

As we can see, at the time of this study, the majority of bots do respect robots.txt when they claim they do. Perplexity's bots do not, but they are upfront about it.



Warning

An important point to take into account when analysing traffic logs is that bots can lie about who they are, especially through their UA. The UA might indicate that the bot belong to Google, for example, but falsifying the UA is trivial. To mitigate impersonation, reverse DNS lookup can be done to see if the IP truly belongs to Google [7].

We also note that some of the bots, like Google's, use the suffix "-extended". This appears when a same bot is used for both indexing and AI features. The benign features are announced as `< bot_name >` while the more controversial AI features are announced as `< bot_name > -extended`, allowing the webmasters, through robots.txt, to block specific parts of the bot's features if desired [63] [69].

User Agent	Category	Company	Publish IP	Claim Respect	Respect in Practice
Amazonbot	AI Search	Amazon	Yes	Yes	Yes
AI2Bot	AI Data	Ai2	No	-	-
anthropic-ai	Undocumented AI	Anthropic	No	-	-
Applebot	AI Search	Apple	Yes	Yes	Yes
Applebot-Extended*	AI Data	Apple	-	Yes	-
Bytespider	AI Data	ByteDance	No	-	No
CCBot	AI Data	Common Crawl	Yes	Yes	Yes
ChatGPT-User	AI Assistant	OpenAI	Yes	Yes	Yes
Claude-Web	Undocumented AI	Anthropic	No	-	-
ClaudeBot	AI Data	Anthropic	No	Yes	Yes
cohere-ai	Undocumented AI	Cohere	No	-	-
Diffbot	AI Data	Diffbot	No	-	-
FacebookBot	AI Data	Meta	Yes	Yes	-
Google-Extended*	AI Data	Google	-	Yes	-
GPTBot	AI Data	OpenAI	Yes	Yes	Yes
Kangaroo Bot	AI Data	Kangaroo LLM	No	Yes	-
Meta-ExternalAgent	AI Data	Meta	Yes	-	Yes
Meta-ExternalFetcher	AI Assistant	Meta	Yes	No	-
OAI-SearchBot	AI Search	OpenAI	Yes	Yes	-
omgili	AI Data	Webz.io	No	Yes	-
PerplexityBot	AI Search	Perplexity	No	Yes	-
PerplexityUser	AI Assistant	Perplexity	No	No	No
TimpiBot	AI Data	Timpi	No	-	-
Webzio-Extended*	AI Data	Webz.io	-	Yes	-
YouBot	AI Search	You.com	No	-	-

Figure 2.3: Tests of companies’ claims about their respect of robots.txt.

2.1.3 The How?

Common crawling methods include requesting the HTML page and parsing the raw result or using browser automation tools such as Selenium [26] or Puppeteer [21]. The later are more complex and resource-intensive but allow to use JavaScript (JS) and the other features of real browsers. This means that the requests to the websites will also look more human-like.

In the case of utilizing a real browser, a headless version might be used. Headless browsers are browsers without a Graphical User Interface. Many popular web browsers have a headless version. They are able to render HTML, dynamic content, style sheets and execute JS just like a normal browser, but they are executed via a command-line interface or using network communication. This is useful because some websites require the execution of JS to display some or all their content. Since they are less resource-intensive than normal browsers and are supported by automation tools, they are often chosen by more advanced crawlers. However, the majority of crawlers are basic ones and cannot, or choose not to, execute JS. According to the data collected through honeysites by [7], only 0.63% of bot sessions supported JS and 0.59% returned a JS-based browser fingerprint.

Multiple research teams were also able to learn more about the behavior of webcrawlers using honeysites and access logs analysis. In the honeysites of Paper [7], 44.9% of bot sessions consisted of only a single request while another 46% of sessions included between 2-20 requests. 58.1% of the bots left within 3 seconds, and among these, 89.5% left after less than a second. However, 10.7% spent more than 30 minutes on their honeysites. In addition, 14.28% of all IP addresses that visited their

honeysites sent requests with two or more user agents. Many of these behaviors will get observed and confirmed during our own research in the next chapter.

The team behind Paper [55] analysed server logs from academic sites and observed that the GET method is used in nearly all HTTP calls. The resources that receive the overwhelming majority (90%) of requests are text (text/plain, text/html) and image files (image/jpeg, image/gif, etc.). The log analysis of an academic web server in Paper [56] adds that Web-related files (html, hmt, asp, jsp, php, js, css) are also sought after.

2.2 Analysis of protection mechanisms

In this section, we will review the state of the art detection, mitigation and protection mechanisms against webcrawlers. we will explain how they work, what their strengths and weaknesses are and the countermeasures bots develop against them.

2.2.1 Content Walling



Definition

We define as **Content Walling** the methods used to hide the content behind some condition, the "wall", that restricts who is able to access the content based on attributes.

The type of content walling that has entered the everyday language the most is the "Paywall". Content only accessible behind a paywall requires the user to pay in order to view it. While this would filter out all automated webcrawlers, this has obviously a huge impact on the website's human users. Less radical forms of content walling exist, like Account Walling, which describes making content only accessible to users with a registered account on the website. While basic automated crawlers that navigate the web will be unable to view this content, bot developers specifically targeting your website will have little to no issue bypassing that wall, as the only extra step needed is for a human to manually create an account, making the content accessible, and send the bots afterwards. Web admins might try to make it harder by protecting account connection with other bot defenses or 2FA but this requires to disconnect the user from their account often, which impact humans users [75]. To avoid this, websites allow the user to stay connected for some time through the use of cookies. Highly practical but it also allows bots to connect to their associated account by sending the cookie [74] [75]. Content walling will therefore not constitute protection against targeted crawling campaigns.

2.2.2 Protocol Standards

Robots.txt

The **Robot Exclusion Protocol** [61] is a standard based on voluntary compliance (non-enforceable) used by websites to indicate to visiting web bots which parts of

the website they are allowed to visit or not. It is deployed through the addition of a robots.txt file at the root of the website. It was originally created to mitigate server overload and is now used to manage content access. It has risen in popularity even more as website owners wish to prevent AI crawlers to collect their site's data for their datasets [63].

Here is an example showcasing robots.txt syntax.

Listing 2.5: Example of robots.txt file

```
User-agent: Googlebot
Allow: /

User-agent: ChatGPT-User
User-agent: Google-Extended
Disallow: /

User-agent: *
Disallow: /private/
```

It works by associating a User Agent with a rule that acts recursively on a directory. In this example, Googlebot is allowed to navigate the whole site (/ is for the root directory) while ChatGPT-User is not. The rest of the bots are only disallowed to enter the */private/* directory and its children directories. We also note that the AI features of Googlebot, described by Google-Extended, are disallowed.



Definition

User-Agent (UA): header part of the HTTP request. It includes information on the machine of the user making the request like its OS, browser, device,... and associated version numbers [75].

As we saw in 2.3, big companies mostly abide by robots.txt. However, malicious bots can just choose not to follow the rules or simply not request robots.txt at all. According to the data collected through honeysites in Paper [7], only 2.2% of bots actually requested the robots.txt file. This is something we will observe in our own research in the next chapter. Let's also note that if a bot detection method flags a user as bot and this bot does not check the robots.txt, it can be flagged as malicious, which is useful to do if we want to apply different defense measures depending on the nature of the bot [57]. Some malicious bots might also check robots.txt with the goal of crawling the pages that the website owner specifically does not want to (e.g. */private/*). We can leverage this behavior against them by disallowing the access to a hidden */private/* directory acting as a honeypot. As neither humans nor bots could find it by simply navigating the site, the only bots reaching it would be malicious ones, which could then be block. One research team goes further [7] and generates passwords.txt files whose path encodes the IP adress of the agent requesting that file. The reason being that some bots will cache the robots.txt file or add the passwords.txt link to their crawl frontier and let future worker bots crawl

the file. Since these bots will have different IPs, a bot with IP 1.2.3.4 requesting a password.txt having been generated for a bot with IP 5.6.7.8 will prove that they are part of a same bot farm, therefore uncovering a common bot operator behind two seemingly unrelated requests.

Despite these more advanced possibilities, robots.txt alone will not be enough of a protection.

AI.txt

With the boom of AI crawlers for dataset data gathering, AI search agents and AI summary features, the idea of having a protocol standard specific for handling AI bots has pop out. Just like robots.txt, AI.txt, introduced by Spawning AI, is put at the site's root but unlike the former, it is meant to be read when an AI model attempts to download media, enabling real-time updates to preferences, even for previously collected data. Its goal is to selectively restricts or permits access to the site's content for AI training [67]. The reason it is made to be read at download time and not when entering the site is because many AI trainers share datasets in the form of a list of download links. The original crawler might have interacted with robots.txt but it will not be the case for the subsequent AI trainers. Below is a template of an AI.txt file. It is similar to robots.txt except each rules applies to a specific file or file format that could be found on the website.

Listing 2.6: AI.txt example

```
User-Agent: *
Disallow: *.aac
Disallow: *.aiff
Disallow: *.amr
...
Allow: *.svg
Allow: *.tif
Allow: *.tiff
Allow: *.webp
Allow: /
```

While it sounds good, it is just another non-enforceable standard like robots.txt, despite its creators arguing it offers a legally enforceable standard, referencing the EU TDM Article 4 exception [63] [19]. Spawning AI aimed at using their position - having a network of AI researchers and partners - to promote the adoption of their standard. Unfortunately, at the time of writing, no significant AI company has chosen to comply with it.

Another team of researchers has made an attempt at creating a AI.txt standard inspired by robots.txt but it has generated even less discussion [1].

Llms.txt

In the same idea, there was a proposal [58] for `llms.txt` and `llms-full.txt` files. While the main idea was to boost SEO/GEO, these file standards would also mitigate server overload by redirecting the bots on those two files. The concept of `llms.txt` is to have a file providing bots with the paths towards the most relevant information on the website. Instead of crawling everything to find their links of interest, AI agents would simply follow the `llms.txt`'s directions towards *page_name.html*.md markdown files containing in plain text the content of *page_name.html* . Meanwhile, `llms-full.txt` would be a file containing the entirety of the website's data in simple markdown format. Bots would only request this file and not the rest of the website and the owner could omit to include in it the data they really want to keep away from the bots. Listing 2.7 represents a mock example of a `llms.txt` file.

Listing 2.7: Mock example of `llms.txt` from `llmstxt.org`'s official documentation [58]

```
# Title

> Optional description goes here

Optional details go here

## Section name

- [Link title](https://link_url): Optional link details

## Optional

- [Link title](https://link_url)
```

Unfortunately, at the time of writing, it has known the same fate as its `AI.txt` counterpart. The proposal has not been endorsed by any major crawler actor and Google has even declared they would not support it. One researcher at Adobe [64] has performed an audit of 30 days of CDN logs for 1,000 Adobe Experience Manager domains to see if the file was requested. If we ignore the indexing bots who probe everything, requests from LLMs-related crawlers were almost non-existent ; only SEO tools showed significant interest.

2.2.3 Reputation Lists



Definition

Reputation List: system that maintains whitelists and/or blacklists of user/network attributes with known malicious participation in sending out bots, in order to block or throttle requests from these flagged actors.

Name	Service
Project Honeypot [17]	IP list of crawler, spammers and other bad actors
Udger [31]	Crawler IP & UA lists
AlienVault [10]	IP list and other security services
BadIPs [36]	IP list of vulnerability scans, spams and DDOS attackers
Blocklist.de [49]	IP list of malicious bots
BotScout [2]	list of bot names, emails and IPs
KnownAgents [30]	UA list of crawlers

Table 2.2: List of Reputation List services used by researchers in this section

One way to protect against unwanted crawlers is to block them when they are detected. It would save a lot of resources if websites could share intelligence on which user is a bot or not. This is where reputation lists come in. We list a few well known ones in Table 2.2. The most popular form is IP blacklisting but UA-based lists also exist. Many different entities manage lists with different purposes, and advanced lists update their databases dynamically using crowd-sourced intelligence, honeypots and real-time threat detection [74] [79] [66] [63]. Unfortunately, these "blocklist as a services" suffer from a big disadvantage: only already detected bots can be blocked. According to an analysis of 100 dedicated honeysites and 9 popular IP blocklists [7], only 13% of bots captured in the honeypots had their IP appearing in these blocklists. This can be explained by the fact that a big part of bots (64.37%) have their IP located in residential IP spaces, not datacenters, or use rotating proxies. In addition, as previously explained, most bots do not stay long enough on a site to draw suspicion, especially in case of Distributed Crawling.

2.2.4 Honeypots

One of the ways Reputation List services can build their list is to deploy honeypots. Whether based on their behavior or attributes, detecting bots is hard, especially since many of them do not stay on the site for long. So what if there were places at which we would be sure to only find bots? This is where honeypots come in. By monitoring honeypots, it is possible to easily detect and take action against the malicious actors falling into them. With in principle, if correctly configured and deployed, no false positives.



Definitions

Honeypot: security mechanism designed to make it easier to detect someone or something that is using a system without authorization [66]. It is a place, resource or action that only illegitimate actors would find themselves interacting with. If the honeypot is an entire website, it is called an **Honeysite**.

- **Passive:** simply monitors the traffic.
- **Active:** attempts to trap the malicious users.

Spider Trap / Tar Pit: active honeypot that traps crawler bots into an infinite labyrinth of pages [66].

Poison Pit: Tar Pit aimed at AI scrappers. The endless pages of the trap contain randomly generated garbage text to poison the dataset that the scrapping bot is feeding.

Deploying honeysites are an effective way to do research on bots since we know the traffic coming to it will not contain any legitimate human users [7].

For normal website owners, integrating honeypots is also possible. The basic idea is to have pages that no normal human would be able to reach, only bots. To that end, the most straightforward method is to place hidden links inside normal web pages that lead to the honeypot pages. Researchers have found many ways to do so.

1. **Simple invisible links.** Using CSS formatting, it is possible to hide elements on the page, like links. Bots generally only look at the page's source code and are unlikely to check the formatting. However, more advanced bots can make use of algorithms and JS to detect such formatting tricks that hide content.
2. **Shadow Links.** Decoy links positioned under a valid human-readable link, at the same x, y coordinates as the valid link and containing the same text, but leading to the honeypot. It means that the decoy link does not have to be invisible, thwarting the bots' countermeasure of detecting invisible links [52].
3. **Multi-links.** Multiple decoy links are placed off bounds of the visible page with text mimicking that of the valid links. Since during Focused Crawling, many bots check the link's name to evaluate whether it fits what they are after, using the same text as valid links instead of */honeypot* ensures to fool more bots [52].

There are multiple formatting tricks to hide content, but to keep the low false positive rate, it is important to take visually impaired human users who use web reader software to navigate the pages. One master student from the naval postgraduate school of Monterey [66] has discovered these programs might be able to read some hidden links, as shown in Table 2.3. The researchers at the University of Georgia had not accounted for that and unfortunately, the Shadow Links and Multi-links

Hiding technique	Display Effect	Webreader-proof
display:none	Element is removed from the page flow and hidden; the space it occupied is collapsed	YES
height: 0; width:0; overflow: hidden;	Element is collapsed and contents are hidden	YES
text-indent: -999em;	Contents are shifted off-screen and hidden from view, but indentation might not completely hide content on some screens	NO
position: absolute; left: 999em;	Content is removed screen view to the lefthand edge ;	NO
position:absolute; overflow: hidden; clip: rect(0 0 0 0); height: 1px; width:1px; margin: 1px; padding: 0; border: 0;	Content is clipped and collapsed. No effect on overflow.	NO
z:index: -1	Content is placed behind elements with greater index number. Hides element by visual obstruction within flow of page.	NO

Table 2.3: Ways to hide elements using CSS

rely on such positioning and z-indexing techniques that webreaders will still be able to read, and thus access the honeypot content.

As discussed in the robots.txt section, this file can also be used for honeypots, as it is not checked by normal humans. Disallowing a specific honeypot page effectively create a honeypot that filters malicious bots from benign ones (who will not check the disallowed page). It also helps mitigate a drawback of honeypots, which is that legitimate web crawlers (e.g. indexing bots) can be fooled by them as well. Since these bots abide by robots.txt, disallowing the hidden honeypot links will filter them out [66].

In addition to links, other elements can be used as honeypots like hidden email addresses to filter out spammers, hidden form when making an account that only a bot would see and fill, to refuse them account creation,... [74]

If detecting and blocking bots is not enough, spider traps are another possibility. They can be created in several ways, such as by adding links with infinitely deep directory trees, or by using dynamic pages with unbounded parameters [66].

Research has found honeypots very effective at capturing bots and results improves as the number of decoy links increases on the pages [52] [74]. Not only that, this detection method is also hassle-free for human users who are never even aware of the honeypots. However, it cannot serve as the sole detection method, especially in cases where the owners want to block scraping in a minimal timeframe as soon

as the bot enters the site. Since bots need to follow the links, it means that they need to be able to stay navigating on the site. They might follow other links first or scrape content before they start following links. Because of Distributed Crawling, one bot falling into a trap will also not affect the others. More defenses are thus to be deployed alongside them.

2.2.5 CAPTCHAS & Challenges

Nowadays, the majority of popular websites use Captchas (Completely Automated Public Turing tests to tell Computers and Humans Apart), at registration or login time for example. With its recognizable "click to prove you are not a robot" caption, it has become the most famous anti-bot measure in the collective eye. Nearly all sites with Captchas employ Google's reCaptcha, which appears to be good at preventing basic bots from accessing a website, while some other choose to implement their own, with varying degrees of success [75]. While we will not go over the history and intricacies of each type and brand of Captchas, it is important to know that they might base their assessment on different factors, like fingerprinting, a risk assessment score or cursor movement, which we will see in later sections.

As some of the first Captchas were essentially asking people to label images or text that the technology of the time could not recognize, they were able to be used to further the progress of Machine Learning (ML) at the time. And now, ML and AI has developed so much that they are able to solve Captcha by themselves with high accuracy [74] due to their improved reasoning and image recognition. Some try to look into potential solutions, like allowing users to solve a Captcha once and receive redeemable cryptographic tokens used to bypass future Captchas, preventing both repeated challenges and user tracking [53].

Google strayed away from labeling-based Captchas with reCaptcha v2 and then v3. The former being a checkbox and the later happening completely in the background. Their method is to collect data about the user's setup and behavior. What data? Google keeps the secret. Any information they would give would make it easier for bot developers to bypass reCaptcha.

While the appeal of Captcha was to serve problems that were easy to solve for humans but difficult for bots, due to the progress in technology, the robots have been closing in. Many automated Captcha Solvers exist for a variety of Captchas [68]. We even saw the rise of Captcha Farms [79], where low-cost human labor is used to remotely solve Captchas. When encountering a Captcha during crawling, the bot sends it to the farm where it is then distributed to one of its workers, who solves it and sends back the solution. Furthermore, their impact on the user experience is non-negligible and modern Captchas are a privacy nightmare as they collect and track much information about the user device [70]. Further investigation on the human CAPTCHA performance accross age groups [62] revealed higher error rates and slower response times for older users, which further raises the problem of accessibility.

In conclusion, Captchas are able to hinder basic bots but will not do much against the more advanced ones or serious targeted attacks. It also cannot be put on every

page without annoying human users, risking a potential userbase loss.

Another possible type of challenge are JS and cookie challenges which consist for example of a browser check requiring the execution of a JS function. These challenges being automatically solved by the browser without human intervention, any bot using a real JS-enabled browser will be able to solve it just as well. However, let's remember that a big honeypot research [7] discovered that this is only the case for 0.63% of bots. Requiring the execution of JS is therefore a highly effective way to filter out bots. Some websites already do so by requiring their pages to be loaded through the execution of an embedded JS script, which also blocks tools such as wget and curl, who cannot execute JS [75].

2.2.6 Content Obfuscation

As mentioned back in Table 2.1, a fairly old issue that has become more prevalent today is rival websites taking content from others and reposting it. While in the old days, one would have to find the target website, copy-paste the content and maybe manually alter it to appear less suspicious, with the advent of AI, this process has become completely automated. It is as easy as sending bot agents to look for a target, scrape the content and use an AI rephrasing tool to repost it on a site, social media, or even in video/audio form using generative AI. From the moment generative AI got available to the public, it was not long until a flood of low-effort plagiarized content, nicknamed "slop", appeared all over the internet. In order to avoid having their content scraped and reposted, some website owners decide to obfuscate the data on their pages. The methods used to do so vary in complexity and efficiency.

One possibility is to have an Adaptive User Interface (UI) Rendering. It consists of only serving the content to users who exhibit human-like behavior. It can be as simple as loading images only when the user has scrolled down to reach them or it can monitor mouse movements, keystrokes or be coupled to any other detection method seen in this chapter. Not having all the content available right away not only improves performances but it also prevents basic bots from getting access to what has not loaded yet [74].



Definitions

Cloaking: technique used to deliver different versions of the content to humans and bots.

Lazy Loading: only start loading content incrementally based on user interaction.

Content Fragmentation: fragment the content into smaller parts that are served only upon activation of predefined user-interaction-based triggers.

Markup Randomization: changes dynamically a website's HTML, CSS and classes in a random, time-based manner.

Text obfuscation

Options for text obfuscation include periodically changing the page's layout to confuse XPath/CSSSelectors scrapers looking for specific HTML/CSS elements, which is commonly used by big online marketplaces against price tracker bots [29]. Textual data in need of protection can also be turned into images that the page displays normally, except that the text is written on the images as opposed to being actual text. This can confuse basic bots but authors of a targeted attack will simply switch to using Optical Character Recognition (OCR) to read the text from the pictures.

The biggest problem that obfuscation faces is that many techniques end up undermining Search Engine Optimisation (SEO). If search engine indexing bots are as confused as malicious bots about what the website is about, they will not be able to reference it correctly and it will prevent the website from showing up on search engines. Cloaking, for example, falls into this exact trap and for this reason, has stopped being considered a valid way to protect websites [46].

Even if people are able to find the site, they might not stay for long as huge alterations to the pages' content might negatively affect the user experience. This is especially true when the text is fragmented or replaced by images as it makes it difficult to select it for example. Moreover, as technology evolves, it is far from impossible that more advanced bots become able to un-obfuscate the content. These defense mechanisms becoming then annoying only to real humans, similarly to the evolution of Captchas and its countermeasures.

Google researchers, of all people, have developed an anti-scraping method relying on Markup Randomisation, that they detail in a patent [78]. The process of which is as follow:

1. They split all the content parts into containers called cells, in a predetermined way.
2. Each cell is given a unique random ID and information on its location on the page.
3. The cells are sorted by ID, unstructuring the data, and then sent to the client.

A scraper would not be able to put back together the cells, making their content useless. Meanwhile, web browsers will not have any issue displaying the text, as they care about the cells' location information and not their location in the code (based on their ID). Unfortunately, it only protects against Xpath-based crawlers, CSS-based ones stay unaffected.

This is why Diab & Barhoum [54] have developed their Markup Randomizer. Their goal was to create a protection against both Xpath and CSS based crawlers, which no previous method had achieved. The later is countered by a system that continuously changes the CSS rules and adapt the HTML elements thanks to a mapping dictionary. The former is countered by the random injection of empty invisible tags in the HTML file in a way that changes the Xpaths of the elements to protect. That project also

aimed at providing a defense layer that would still be working even if the other layers were to go down (e.g. Cloudflare outage). Another important thing they mention is that obfuscation techniques should be used after a CSS/HTML optimisation process so that the impact on the file size of adding the obfuscation stays minimal.

Bolbol & Barhoum [50] take inspiration from the above research and make a Markup Randomizer that adds a random number of *span* elements in the page's body, containing hidden random characters and using a randomized class name to prevent automatic deletion. This ensures any reposter will have to manually remove the random characters present in the scraped content if they want to repost it. Because the *span* are hidden, it does not affect the reading experience on the original website.

Paper [51] deploys a Markup Randomizer on the client-side that mitigates the big problems of obfuscation, which reduces server load. They do not obfuscate titles, heading and meta descriptions in order to minimize the impact on SEO. They also couple it with some offensive measures against AI crawlers: integrating prompt injections in hidden *div* elements containing misleading or disruptive instructions, and content overload to waste AI tokens by adding huge amount of useless hidden text to the page. Note that the later should be used with care to not worsen the site's performance. However, their results found that these offensive measures did not affect bots significantly.

The most advanced obfuscation process is proposed by paper [68]. They present a multilayered strategy they call Polymorphic Web Code Generation Engine. It dynamically rewrites the webpage content on each request to obfuscate sensitive elements. At each render, the structure is unique which significantly hinders bots' ability to parse it. 3 layers are used in this approach.

1. **Webpage minification algorithm:** removes non-executable elements like comments, whitespace, and newline characters to reduce payload size and conceals structural cues used for parsing.
2. **Dead code:** dynamically generated Boolean expressions with known outcomes that appear complex to automated analyzers. None of this affect program behavior but it significantly increase control-flow complexity, making reverse engineering and interpretation of the code more computationally expensive and less reliable.
3. **Random sequence of transformation funcs:** use of a pool of 16 JS-based obfuscation functions to protect sensitive client-side data.

Their results show a decrease in successful bot scraping attacks by 92.3% while maintaining a good user experience.

Image obfuscation

Images too have become valuable scraping material since image generation AIs have become widely used. Obfuscating images is harder. Glaze [6] and Nightshade [15] - popular among artists - offer to inject special noise patterns in the picture to

render them unusable for AI training, and even act as poison. However, their actual effectiveness is widely debated and increasing the noise patterns will decrease the image quality. Meanwhile, the "other side" has already started working on de-noisers able to filter out the protective noise. They are also very computationally intensive, which is why the art website Cara, who had adopted the technique, had to stop using it after a while.

Another option inspired by Content Fragmentation is to segment the image into multiple parts and use CSS to reposition them correctly on the page. Scrapers downloading the images would only get the parts and not the full picture, making it useless without manual re-gluing. This is what the very recent Katana project [77] does. It converts the input image into any number of layers. These image layers are useless by themselves as they look like garbage thanks to the application of random operations like blend modes, effects and cuts. The source code of the page only contains the shuffled layers, but they are overlaid at rendering time and by doing so, recreate the original image. In addition, other techniques mentioned above like Markup Randomisation are supported to mess up with XPath-based Scrapers.

Despite being the state of the art in image protection, Katana is still not perfect. Screenshot Bots exists and they can take automated screenshots of the final render of the page or a specific part of it that they can select using Xpath or CSS Selectors.

Conclusions on obfuscation

The existence of Screenshot Bots, like Shot Scraper [27], essentially nullifies the results of the entirety of obfuscation techniques since those bots are able to "see" the exact same thing that human users see. Furthermore, all content relying on a specific obfuscation technique will be rendered completely defenseless once the obfuscation technique gets bypassed or reversed. Glaze might have been very effective at the time of its creation, but any image protected by its initial version is now just as defenseless as images not using it at all. This adds a lot of motivation to scraper developers to find ways to break obfuscation defenses, as they would instantly unlock the access to all the content that was previously protected by them. In a way, it adds a target to that content. We conclude that an obfuscation layer might be useful as an annoyance to scrapers and a last layer of defense when the previous ones fail (as was the point of Diab & Barhoum's project) but that we cannot rely solely on it.

2.2.7 Fingerprinting

Fingerprinting is the name given to the set of techniques that allow to identify a user when they come back to a website or even accross websites, by analysing the characteristics of their setup. For example, if you have a specific font installed, among all the users, only x will also have it installed. It is not enough to uniquely identify you. But out of this set of x users, only y users have the same screen size as you. Only z will have the same browser extensions. By analysing each attribute, it allows to narrow down the set of matching identities until a single one remains. This is how websites can track an unknowing user accross the web.

Fingerprinting can also be used to identify attributes that are normally not found with human users. For example, Headless Chrome does not send HTTP *accept-language* headers, which indicate the language used by the client [79]. Extremely small window sizes, like below 50 pixels, reduce computational overhead and as such, are often used by bots. Analysing the UA can also be considered Fingerprinting. UAs such as "*python-request*" strongly indicate bot activity [70]. Anti-bot scripts attempt to detect web automation tools like PhantomJS, Nightmare, Selenium, and headless Chrome browsers by trying to capture known variables such as *window.callPhantom* (present in PhantomJS and Slime JS) or *navigator.webdriver* (Selenium) [48].

Many Fingerprinting libraries and services exist, like Fingerprintjs2 [4], Plug-inDetect [18], Cover Your Tracks (formerly Panopticlick) [44], FontList.swf [5], Evercookies [23], Modernizr [11], HUMAN Security (formerly PerimeterX) [8] or ThreatMetrix [28].

Most webmasters use one of them, but some write their own custom scripts.

The issue with Fingerprinting tools is twofold. First, web automation tools and them are constantly trying to one-up each others. Many checks done by Fingerprinting tools become outdated as web automation tools update to evade them, as tested by Paper [48]. Secondly, many of the attributes that are checked can be spoofed by the user in some way or another. This is not all bad however, as it allows for a different type of fingerprinting check: inconsistency checks. In the process of spoofing their attributes, malicious actors might make mistakes and create combinations of attributes that cannot be found with normal human users [76]. In other cases, once well-spoofed attributes might become outdated. For example, mismatches in version numbers in the UA can appear if the user does not update their spoofed UA.

Detecting spoofing is not a sufficient reason to label a user as bot or malicious however. As it can be understood from the description of Fingerprinting, it is an extremely privacy invading process. Even more so when knowing that it is used not only to protect from bots, but also just to track people using Fingerprint-based user identities. Many privacy-conscious people choose to resort to the same spoofing tools and techniques as bot developers in order to obscure their setup characteristics and evade Fingerprinting.

2.2.8 Machine Learning

With no single bot detection/protection method being 100% accurate and sufficient, some have turned to Machine Learning (ML) in order to craft detection systems that can take many data points into account, improve with larger datasets and evolve with the continuously adapting bot landscape. A well-trained ML model can assess a user's behavior as it happens and learn from patterns of normal behavior to uncover non-intuitive patterns of malicious behavior [47]. In this section, we will see two types of ML approaches. One based on setup attributes, like a ML version of Fingerprinting. The other based on tracking the way the user interacts with the website. Regardless of the methodology, it is important to say that a ML model's

performance will greatly depend on the size and quality of its training dataset and that ML classification can be resource-intensive.



Definitions

Session: sequence of requests with the same IP address, the same UA string, and where the time interval between any two subsequent requests does not exceed 30 min.

Attribute-based

In paper [72], researchers design a neural-network-based model to perform real-time session classification, i.e., classification in the shortest possible time. To create their training dataset, they used log data from a real e-commerce website and labelled as bots the requests whose UAs were present in Reputation Lists. Udger's, in particular. Human labels were given to those Udger classified as a normal browser. Considering UA spoofing is easy and widespread, this way of labelling human traffic does not seem reliable to us. One of the most crucial design choice for a ML approach is the choice and crafting of features. This paper builds the following features:

1. The HTTP method.
2. A boolean *is_referrer_empty* on whether a referrer has been provided.
3. 5 boolean features from the URI on whether the requested resource is a page, a data file, a graphic file, a script or a style sheet.
4. The HTTP response status.
5. The response size.
6. The time interval between timestamps of the current request and the preceding one.

Those features have a big advantage: they are readily obtainable from HTTP request headers without intervention by the server, which guarantees easy integration on the website and minimal overhead. This is especially important for scalability since the method requires to analyse every single request.

In the first stage of their approach, each HTTP request is individually assigned a probability of being either a bot or human. In the second stage, a sequential classifier (Wald's Sequential Probability Ratio Test) is used to combine those probabilities and converge to a final decision for the session with the minimum number of observations. One issue that arises from this process is that the number of requests per session might happen to be less than needed for an accurate decision. This is especially true if we take into account the research we've seen previously on Distributed Crawling and the analysis of honeypots [7]. Most bots stay on the website for a very small time frame and send very few requests.

In their results, they were able to classify 99% of sessions within 6 requests, with 16.7% classified at the first stage and 67.5% at the second stage.

Paper [73] attempt a similar approach with the C4.5 decision tree algorithm using 24 features extracted from a session. Among them are some features we have not seen other studies use. They have a feature for nighttime requests, which is interesting as bots can stay active at night, but it could also just be users from another time zone. There are also features for the width and depth of the traversal in the URL space, and whether the session contains multiple IPs or UAs. Their findings were that, even for the indicators that correlate the strongest with bot activity, none of them exceeded 0.5 correlation. Hence, using any of these attributes alone is insufficient for accurate classification. Interestingly, the width feature is positively correlated with robot sessions, as opposed to the depth feature. This confirms the behavior described in the rest of the literature that the breadth-first search is used to retrieve files from websites. They also notice a high amount of multi-IP sessions, which we can correlate to Distributed Crawling, with worker bots that parallelize their crawling operations. Lastly, they observe positive correlation between bot sessions and nighttime requests, but we believe it is because their data comes from their university website’s logs where the users are mostly concentrated in the same timezone as the university and we doubt it would work for websites with a more general userbase. With the results of their classifier, they conclude web robots can be detected with more than 90% accuracy after 4 requests.

Choosing the right features is not the only design decision impacting the performances of the classification, results can vary from one ML model to another. Paper [47] tests many different models such as Support Vector Machine (SVM), K-Nearest Neighbor (KNN), Decision Trees (DT), AdaBoost, and Bagging. They found KNN was the best fitted for their use-case, but the effectiveness of a model may depend from the features and dataset used, not just the goal to achieve.

Another approach is paper [80], using federated deep learning. And lastly, paper [40] with the use of a Bayesian network classifier, a probabilistic model. They use a smaller amount of session-based features but we note one we have not seen before:

They calculate the maximum sustained click rate, meaning the maximum number of HTML requests achieved within a certain time-window.

Dynamics-based

Paper [70] explores the possibility of analysing mouse patterns to distinguish bots from humans and integrate this technique with HTTP request features. This is an improvement on the previous approaches, as mouse trajectories and general interaction behavior is much more difficult to replicate than simply spoofing the request fields. While basic bots will not go as far as trying to replicate mouse movement, this would be a great detection tool against targeted crawling. The set of features they use is extracted from multiple requests inside a same session.

1. The fraction of HTTP requests that result in 4xx errors.
2. The fraction of requests for CSS files.
3. The fraction of requests for JS files.
4. The fraction of requested URLs that contain the previously requested URL as a substring.
5. The total session time.
6. The number of “/” characters in the URL path.
7. The mean and standard deviation of intervals between requests.

For the mouse movement features, they extract the coordinates, path length, tangential & angular velocity, acceleration, straightness, jitter and jerk, as well as the action duration, mean, standard deviation, minimum, maximum, and range for each of these features.

They tested their ML classifier against humans and bots using ghost-cursor [35] to replicate realistic mouse movement. While they achieve good results, they lacked real-world data. They also note that bots could save and replay authentic human movements to avoid detection.

For completeness’ sake, we also note that a purely mathematical and non-ML version of mouse dynamics analysis had already been explored in Paper [65] using advanced vector analysis and fractional derivatives.

In their second approach, they use keystroke dynamics. Features include hold latency, press latency, release latency and inter-key latency. Trying both key-dependent and key-independent modelling, their results show key-dependent modelling is better at distinguishing bots from humans.

2.2.9 Content Delivery Networks



Definitions

Content Delivery Network (CDN): geographically distributed group of servers that caches content close to end users and allows for the quick transfer of assets needed for loading internet content.

CDNs are not just services to reduce loading time, they have many security capabilities due to their ability to apply traffic filtering and rate limiting at a scale. CDNs analyze incoming requests in real time to perform Fingerprinting and other previously seen techniques, and find patterns indicative of bot behavior. Malicious requests are then filtered out before reaching the server. In addition, many also have built-in bot-related features like Captchas and JS verification. The big advantage of CDNs and other types of Software as a Service (SaaS) platforms is that they can be easily integrated within an organization’s infrastructure and provide real-time insight

and analytics about traffic patterns and vulnerabilities. With their many setting customisation possibilities, the customer can tune the system to their own use-case.

The biggest CDNs, Cloudflare and Akamai, even use ML that dynamically adapts to evolving bot tactics. Those currently stand as the most efficient and widely used defense against all kinds of web bots, from basic ones to sophisticated threats [74].

Of course, such SaaS are not free and no open-source project has come close to their scale. Their scale clearly being their biggest power. With the billions upon billions of requests going through Cloudflare servers every day [41], they are able to build a large and strong dataset, which explains their position at the top. Unfortunately, at the time of writing, details on their methods have not been published.

2.3 After the detection

It is not enough to detect incoming bots, we need to enforce some policies against them to neutralize them. In this section, we will see how to retaliate against web crawlers.

2.3.1 Bans and limitations

The most straightforward way is to block their IP. Adding rules to the firewall is easy and effective. But only if they are alone, which as we have seen, is often not the case. When a bot is banned, others may be sent or it may come back through a proxy. Additionally, some bots have the ability to notice when they have been banned and infer what made them get banned. For example, if the bot clicks on a honeypot and instantly gets blocked. It might understand the link was a trap and warn others to avoid it [75]. The same goes for rate limiting. If a bot notices it is being rate limited, it will understand it has been found out and will automatically switch to a slower crawling speed to avoid triggering more alarms. Then, the next bots replicating the same behavior will be harder to detect. When going for blocking or limiting, it is recommended to wait a bit between the moment you detect the bot and the moment you block it so that it has a harder time identifying what caused it to be detected.

2.3.2 Aggressive Methods

With the growing aversion towards web crawlers and their impact on the web, some people have developed more aggressive methods of dealing with them. Paper [75] describes a JS script that generate endless rows of text until the browser crashes, another one that call a same JS function endlessly, causing a stack overflow, or a CSS file containing a single so extremely long line that it crashes the browser when it tries to load it. Avoiding this kind of traps is another reason most bots avoid executing JS when crawling. Fortunately or unfortunately, modern browsers are able to handle these malicious scripts gracefully without crashing. Bots using real browsers will therefore be mostly unharmed.

Instead of destructive measures, building traps is also possible. For example, trapping the crawler in an endless loop between a pair of pages, each redirecting to the other. In the section on Honeypots, we had also mentioned Tar Pits. Cloudflare has recently added a special kind of trap as one of their anti-bot features [41]. They use generative AI to create labyrinths of pages full of text (but still convincing human-like content) on various topics. Conversely to Poison Pits whose goal is to poison AI scrappers with garbage content, the text in Cloudflare’s maze is made of fully accurate information, just unrelated to the rest of the website. Their goal was only to trap bots in a convincing maze to waste their resources and not to spread more misinformation. They have not made the details of their whole process public but they claim to use a custom HTML transformation process to seamlessly integrate doors to this pre-generated content maze as hidden links on existing pages without disrupting their original structure. They even go further by using meta directives (no details are provided) to prevent search engine indexing and thus preserve SEO. This is important because allowing the good crawling bots, especially indexers, is a crucial requirement that makes or breaks other defense systems as we have mentioned before. As an effort to mitigate another big concern, they claim to be ensuring that trap links remain invisible to human visitors thanks to special styling but also by only presenting these links to suspected AI scrapers.

When creating such negatively impacting systems as we’ve seen in this section, making sure no human falls into those traps is absolutely vital. It is not something the other trap-like defenses were doing. But ensuring no human falls into the labyrinth has an even more positive consequence: if all traffic in the trap is from bots, then the trap effectively doubles as a Honeypot. Cloudflare writes on their blog that they use the trapped traffic to train and strengthen their ML detection models.

2.4 Review summary

In this review of the state of the art, we have learned what the web crawling landscape is composed of and seen many detection, protection and retaliation measures against bots. Despite the abundance of research in this area of cybersecurity, no defense has yet been found that could protect a website from 100% of bots while maintaining a good user experience for humans. We have seen that by combining defenses together, they could fill some of each others’ gaps and give better results. As a summary, Table 2.4 contains all the types of defense mechanisms we have read about, with their advantages and disadvantages.

This review of the literature allows us to see that each approach has an impact on different things. We conclude that these defense methods will be more or less effective and desirable depending on the precise use-case and goals of the website owner. In particular, what they want to protect, what resources they have, the trade-offs they can afford, their stance on web crawling and AI, their budget and technical skills. Understanding those requirements will be necessary to design a fitting defense system for our own case.

Defense	Effectiveness against simple bots	Effectiveness against targetted attacks	Impact on humans	Implementation complexity	Easiness to bypass
Content Walling	Very high	Low	High if not already present	Big changes needed to the website	Needs human intervention
Robots.txt	Only if they abide	None	None	None	Compliance not mandatory
Reputation Lists	Only if in the list	Only if in the list	None	Low	Just changing the IP (easy for bot farms)
Honeypots	High, but delayed	Lower if bots can detect them	None if done correctly	Low	Needs detecting capabilities
Captchas	Medium	Low	Annoying	Low	Bypass by adapting behavior
Content Fragmentation	High	Low	Medium	High	Needs to reverse the obfuscation
Markup Randomisation	High	Low	None	High	Needs to reverse the obfuscation
Fingerprinting	Medium	Low	Low	Low	Adapt behavior + spoof attributes
Attribute-based ML	High	Variant	Low, ideally	High	Hard (adapt behavior/attributes)
Dynamic-based ML	High	Variant	Low, ideally	High	Very hard (adapt behavior/attributes)
CDNs	High	Medium	Variant	Easy but potentially costly	Very hard, they are blackboxes

Table 2.4: Summary of defenses against bots

Chapter 3

Methodology & Implementation

In this chapter, we will describe the design and implementation of our approach to the bot traffic problem. First, we will establish requirements based on our use-case in order to design the best fitting solution. Then, we will go through each step of the implementation process, from data collection to the final code, explaining our decisions and thought process.

3.1 Design Requirements

In the previous chapter, we have seen the advantages and disadvantages of the state of the art bot protection techniques and that not all of them fit all the use-cases. To understand our own, we must remember our problem statement.

1. The UCL network has its server overloaded because of the high bot traffic.
2. Most bots should be filtered out to decrease server load.
3. SEO must not be affected
4. Scraping is not an issue we need to defend against
5. Impact on human user experience must be minimal
6. The solution should not use a third-party paid service

Considering that the bot technology is in constant evolution and that most individual defenses have already countermeasures to them, we believe that a ML-based defense system would be the most appropriate. ML models can find patterns humans cannot and adapt to new ones, which is necessary if we want our solution to be relatively long lasting.

There are three important steps to create a ML system:

1. The dataset
2. The features
3. The model

We first needed to build a dataset. This was especially important because we knew that the effectiveness of the ML is greatly influenced by the size of the dataset. A reason that made Cloudflare's bot protection so powerful was their giant dataset. Unfortunately, we did not have such a dataset at hand. We would need to collect

data ourselves. Like Cloudflare [41] and Paper [7], we believed the best way to do so was deploying Honeybots.

We had learned from the previous research [7] that a large portion of bots stayed on websites only a very small amount of time and sent few requests. In fact, 44.9% only sent one single request. In order to catch bots on-the-fly as fast as possible, we decided to try building a classifier that would take decisions based on a single request instead of an entire session like other systems from the literature. This would prevent us from using many types of features like time intervals or cursor behavior. But would see that by analyzing the data of our Honeybots, we would find new useful data points that could be used as features.

3.1.1 Benchmark

To set ourselves an objective to compare our solution to, we wanted to create a set of hardcoded rules that could identify the most obvious bots using simple but accurate fingerprinting. This rule-based method would serve as a benchmark that our ML-based solution would need to surpass to be considered effective. Before we could know what rules to write, we first had to go forward with the creation of Honeybots and analyse the resulting logs to see how to identify the simple bots.

3.2 Honeybot deployment

The goal of creating Honeybots is to collect data we will be sure to be 100% coming from bots. Of course, traffic logs from the UCL websites already contain both human and bot requests but the problem is that this data is unlabeled, meaning we cannot be sure whether one request is from a bot or a human unless we'd run some detection system, which is the exact thing we are trying to build in the first place.

Some papers [72] [73] built directly their dataset from unlabeled data, by labelling it through a custom process like looking for keywords in the UA or other forms of Fingerprinting. We prefer the Honeybot approach as manually labelling data only allows to collect data on what you can already identify as a bot. It does not take into account the "unknown unknown" or the spoofing of attributes.

Concretely, we will build two honeysites. One with all the doors open to bots, the other with a robots.txt asking the bots to stay away. Doing so, we hope to be able to see a difference in the type of traffic and also test ourselves the claims of the big companies that their bots respect robots.txt.

3.2.1 Honeysite design

The most important thing to keep in mind is that no human must be able to find our honeysites. To this end, we must give our sites a very complex and specific name that should not turn up in web searches unless specifically sought after. For the sites themselves, we want them to be realistic enough as to see how crawlers traverse them.

Listing 3.1: Example of the hugo.toml config file

```

baseURL = 'https://keta9.com/'
languageCode = 'en-us'
title = 'The_Emerald_Semaphore_of_Keta-9'
theme = 'mainroad'
pagination.pagerSize = 10
renderer.unsafe = true
enableRobotsTXT = true

[Params.sidebar]
  home = "right"
  list = "right"
  single = "right"
  widgets = ["recent", "categories", "taglist", "sidemenu"]

```

To that end, we will create wiki-like sites with many interconnected pages sorted in categories, a search feature and a "recent pages" feature. For the sake of testing how AI agents interact with them, the pages' content should have information that is easily querifiable. The context of a wiki is particularly adapted to that but there is an issue. If we fill our site with real information on a specific topic, there is no guarantee that an AI agent would go look at our sites for information. On the other hand, generating easily verifiable misinformation would not fool the AI so we cannot simply invent a new fake plant or historical event and write about it. And that is not even tackling the ethical aspect. Instead, we had the idea to create wikis on fake movies. New movies pop up all the time, some a very obscure and being explicitly fictive allows us to give our websites some very complex and specific names like we wanted, while minimizing the impact of spreading misinformation. Thus were born "The Emerald Semaphore of Keta-9" and "Klyth-77: The Halcyon Codex of Sarn" that we'll respectively shorten to "Keta" and "Klyth" for the rest of this paper.

3.2.2 Honeysite creation

The honeysites had no need to be complex. To create them, we used GoHugo [34]. GoHugo allows to create websites in a very fast and easy way almost only using markdown files. Once a GoHugo theme is downloaded and added to the .toml config file, GoHugo will simply take the markdown files, one per web page, and convert it into HTML to build the website. Adding the special features our wikis require to feel real is made very simple. For example, displaying recently modified pages or listing page categories is done by adding a simple line to the config as shown in Listing 3.1.

For the actual text of the pages, it did not really matter as long as it felt real. We took inspiration from Cloudflare's labyrinth generation [41] and generated content pages using ChatGPT. We divided the content between 3 main categories: "characters", "places" and "others". Each containing a few entries. First, we prompted ChatGPT to generate names of pages that fitted these categories. Once it was done,

we asked it to write a page of information on each, with the special instruction to make references to other pages and include their links. The interconnection of pages would be useful to observe the crawling behavior. In addition, we generated some pages with no links and some that had none linking to them. This was helpful because some crawlers find pages with many links more interesting and we also wanted to see if they would reach a page that was not linked anywhere.

Another thing we wanted to observe was whether media files were requested. On the sites' main pages we added different types of files.

- A photo of an "actor", which was in fact an image generated by Thispersondoesnotexist.com.
- A pdf file titled "Extended Credits".
- A video titled "VFX test", generated by AI.

While we had designed our Honeysites to be near impossible to find by a human, the risk could never be zero. For ethical reasons, we needed to put some sort of disclaimer to warn any lost human that they had stepped into a honeypot.

This is a test website, everything is fake.

This was a nice disclaimer, but a problem arose. How could we communicate it to humans, but not bots? We did not want our disclaimer to have any impact on the crawling of bots, so simply having written at the top of the page that everything here was fake information was not the way to go about it. One idea could have been to write the text in an image and display the image, but many bots have OCR capabilities so it would not have been an optimal solution. To write a disclaimer only readable by humans, we came up with another solution:

1. We cut the sentence into several 4-characters parts.
2. We embed each part in an HTML element.
E.g. `"webs"— >< pclass = "w" style = " — -o : 4" > webs < /p >`
3. We dispersed each part all over the page.
4. We used CSS to modify the absolute position of each part in order to replace them in the correct order at the top of the page.

This way, no bot could possibly put the disclaimer together, unless it had been specifically developed to do so.

3.2.3 Setting up tests

Next, we set up a few things that would give us some up to date insight on crawler behavior for when we would do the log analysis. We used CSS to hide some links on the page to confirm whether invisible links were followed by the crawlers. We

Listing 3.2: CSS code to put back the disclaimer together.

```
/* HTML class example. */
/* --o is a variable which corresponds
to the index of the part in the sentence. */
<p class="w" style="--o:4">webs</p>

<style>
.w {
    position: absolute;
    top: 50px;
    /* Each part is offset according to its offset. */
    left: calc((var(--o)) * 55px);
    display: inline-block;
    margin: 0;
    font-size: 24px;
    font-family: sans-serif;
}
</style>
```

Listing 3.3: CSS code to hide links.

```
/* HTML class example. */
<li class=secret1>[1](cut-content/scene1)</li>

</style>
.secret1 {display: none;}
.secret2 {visibility: hidden;}
.secret3 {height: 0; width:0; overflow: hidden;}
</style>
```

used different methods we saw previously: *display : none*, *visibility : hidden* and *height : 0; width : 0; overflow : hidden;*. Of course, these hidden links would not lead to honeypots because we already were in one. Instead, they just lead to pages about deleted scenes from our fake movies.

In addition to CSS, we made use of JS to add some interesting tests. Since we had learned that most bots do not use JS, we could test whether this would be an effective way of hiding content with a page *production.md* in which the content changed based on whether the user had JS enabled or not. We added a simple `<div id = "js - production - content"></div>` and a call to our script *js - required.js* with `<script src = "/js/js - required.js"></script>` at the bottom of the page. The JS script looks for our *div* and insert our special text inside. Note that the JS code is in a separate file and not in the page's own HTML or else the content of *container.innerHTML* we wanted to hide would be returned as the bot requested the HTML page. The last thing we needed was a fallback text for when the user did not have JS enabled. We could write it inside a `<noscript>`

Listing 3.4: JS code to insert content in a page.

```
document.addEventListener("DOMContentLoaded", function () {
  var container = document.getElementById("js-production-content");
  if (!container) return;

  container.innerHTML = '
    // insert page content here
  ';
});
```

Listing 3.5: JS code to insert content in a page.

```
<div id="js-awards"></div>

<noscript>
  <meta http-equiv="refresh" content="0;
    url=/awards-simplified/">
</noscript>

<script src="/js/js-awards.js"></script>
```

element, which would only be displayed if JS was disabled.

Another way to go about it was to automatically redirect users without JS to another page, possibly a simplified version of the current one. The code of our *awards.md* page is shown in Listing 3.2.3. It includes the same JS-based text insertion as above but the content of `<noscript>` differs. Here, it triggers an automatic redirection to a different page.

To finish up our Honeysites, we added a few files to the root directory: *robots.txt*, *ai.txt*, *llms.txt* and *llms-full.txt*. This was the occasion to see for ourselves whether the crawler bots were abiding by those files' guidelines or not.

For the *ai.txt*, we used the *ai.txt* generator of <https://EmpathyFirstMedia.com> to allow everything except image and code file requests. What was allowed was of no importance for us. What mattered was to see if there was a difference of behavior for what was allowed and not allowed. If the file was requested at all, that is.

For *llms.txt*, we followed the standard's example and create a markdown file starting by a brief description of the website followed by a listing of the important pages.

For *llms.txt*, we simply copy-pasted the entirety of the site's content into the file. Since we were using GoHugo, all the content was already in markdown.

Listing 3.6: Keta's robots.txt

```
$User-agent: *$ \\
$Disallow: $
```

Listing 3.7: Klyth’s robots.txt

```
$User-agent: *$ \\  
$Disallow: /$
```

3.2.4 Deployment & Monitoring

To avoid showing any relationship with our University, two domain names were purchased, keta9.com and klyth77.com, and the sites were published. Github Actions were written to auto-deploy the sites when new commits were pushed.

To capture traffic data, we used Caddy [71]. Caddy is an open-source web server with many features and utilisations. For our current and later purposes, its HTTP server, logging and TLS Termination Proxy capabilities were what we were after. SSH was used to connect to each honeysite’s virtual machine to access the log files. We then waited two weeks in order to collect enough data to start our analysis.

3.3 Traffic log analysis

After two weeks of running the honeysites, we did a first collection of the traffic data. We already had 7000+ requests in the Caddy logs and we kept the honeysites going to collect more in the following months.

3.3.1 General statistics

We started by writing a python script to parse the caddy log file and extract data we were interested in. Starting with statistics about the requests, Table 3.1 shows the percentages of each type of request.

We also noted that there had been no requests for ai.txt, llms.txt nor llms-full.txt. As we were expecting, those standards were not used in practice.

A second thing we wanted to check was whether the bots from big companies still respected robots.txt like they did at the time of Paper [63]. Our logs showed this information was still accurate. To compare to Klyth, whose robots.txt disallowed bots, we notice bots from OpenAI (OAI-SearchBot), Google (Googlebot) and Facebook were indeed requesting robots.txt and stopping there. However, despite robots.txt, the log files of both honeysites were very similar. This was because the majority of bots did not check the file or did not stop at it. The rest of our analysis will make us understand why.

Thales reported that 44% of bot attacks targeted APIs [38] but in our logs, only 1.85% of malicious requests were related to APIs. Likely because our logs did not contain targeted attack samples.

Type	Percent of requests
Cloudflare Workers	42.47%
Bots found from UA	27.6%
Vulnerability scans	66.57%
Cloudflare Workers	63.8%
API attacks	1.85%
Total bots found using rules	89%

Table 3.1: Types of bots in the logs of Keta

Listing 3.8: List of bot UA tokens.

```
"gpt", "CMS", "curl", "openai", "ai", "google", "bing",
"Perplexity", "Claude", "bot", "Headless", "Go-http-client",
"aiohttp", "Python", "python", "MJ12", "CensysInspect",
"okhttp", "DotBot", "Preserving", "localai", "OAI",
"BulkSiteChecker", "SkypeUriPreview", "undici", "Scrapy",
"wget", "+https", "leakix", "NetcraftSurveyAgent",
"facebookexternalhit", "letsencrypt"
```

3.3.2 Deeper analysis

User Agents

To continue, we started to manually analyse the logs looking for trends, patterns and what the bots were doing. We began with the UAs. They are in many cases the first sign of the presence of a bot. Some UA strings are very telling, like any bot having "bot" in its name. Automated requests also come with their own UA strings, like "python", "curl", "wget". We noticed many crawlers link their owner's website in the UA to showcase who they are and why they are crawling. We believe this might be a way to show they are not malicious and have more chances to not get blocked if the webmaster think their purpose is legitimate. The full list of non-human UA strings is shown in Listing 3.8.

URIs

Another way to instantly tell that we are dealing with a bot is through the requested URI. We found out the vast majority of requests were coming from vulnerability scanners. Bots that go from one website to another trying to see if they can find any vulnerable resource that could be used to attack the site. It translated in the logs by requests for pages our Honeypots did not have. For example, they were looking for admin, config, API, debug, test, database and critical information pages and scanning to see the type of technology the sites were using.

To give some varied examples : `/admin/config` , `/.git/logs/HEAD` , `/dev/.env` , `/__profiler/phpinfo/phpinfo.php` , `/env.backup` , `/xampp/phpinfo.php` `/laravel/info.php` , `/mail/.env` , `/nginx/.env` , `/env.backup` , `/api/shared/config.env` , `/service/email_service.py` , `/server/config/database.js` , `/.DS_Store` , `/__all_dbs` , `/login.action` , `/server - status` , `/config/storage.yml` , `/files/` ,

/ecp/Current/exporttool/microsoft.exchange.ediscovery.exporttool.application, /s/537313e28373e2430313e2033313/_/;/META-INF/maven/com.atlassian.jira/jira-webapp-dist/pom.properties , /application.properties , /swagger.json , /app.py , /apis/controllers/users.js.

The majority of malicious request were scanning for wordpress pages to exploit like */wordpress/wp-admin/setup-config.php*, */wp-content/plugins/pwnd/cloud.php* or *xmlrpc.php?rsd* and some were for AWS like */aws/credentials*. Each coming in different variations (e.g. */shop/wp-includes/wlwmanifest.xml*, */wp2/wp-includes/wlwmanifest.xml*, */2019/wp-includes/wlwmanifest.xml*), testing each possible place for the specific file they were looking for. 66.57% percent of the requests were vulnerability scans. This overwhelming proportion of malicious bots explains why the Klyth honeypot had its robots.txt disregarded. This partly explains the observation that error status codes, especially 404, have a high correlation with bot traffic [55].

Code 404 (NOT FOUND) is not the only error code we had noticed the abundant presence of, code 308 (PERMANENT REDIRECTION) was also all over the logs. The recurrent pattern was a first request to a valid page (e.g. */others/kestrel-clockwork-sequence*) resulting in a code 308, followed by a request to the same page but with a */* at the end (*/others/kestrel-clockwork-sequence/*). Our first interpretation was that bots requested the URL without */* and were "redirected" to the one with the */*. However, the RFC 7538, which describes the role of the code 308, directed us to a better explanation. The RFC Series contains specifications and policy documents for the internet, mainly produced by the Internet Engineering Task Force, the Internet Research Task Force and the Internet Architecture Board. Code 308 indicates that the page had been assigned a new permanent URI different from the one it was queried with. This status code is similar to code 301 (Moved Permanently) except that it does not allow changing the request method from POST to GET. And it can be used to automatically redirect HTTP traffic to a HTTPS version. In conclusion, it meant that bots were trying to access the honeysite with HTTP and were redirected to HTTPS.

Cloudflare Workers

After that, we started investigating the rest of the fields in the requests. We noticed a large amount of malicious bots had a specific series of fields: *Cf-Connecting-Ip*, *Cf-Ray*, *Cf-Worker*, *Cf-Ew-Via* and *Cf-Visitor*. After some research, we found out they were fields added by Cloudflare to requests made by their Cloudflare Worker (CFW) bots. Looking back at 3.1, it seemed a significant majority of vulnerability scanning we had observed in the last subsection originated from users of CFWs (63.8%).



Definitions

Cloudflare Workers: serverless platform for building, deploying, and scaling apps across Cloudflare's global network [43].

Users of CFWs can build and deploy full-stack apps able to manage HTTP requests that runs on Cloudflare's edge servers. Bad actors using CFW's Fetch API to send HTTP requests has been a known security issue for a while [42] but it appears no solution has come from Cloudflare yet. Exploiting CFWs allows attackers to send malicious requests directly using Cloudflare's own IP addresses. As a result, websites using Cloudflare for their security can still be victims of this type of attack, as the IP is by default on Cloudflare's whitelist. Cloudflare publishes the IP ranges that belong to them so we were able to verify that IP we found in the logs did belong to them.

Reading, the Cloudflare documentation [39], we see that more fields were actually added by Cloudflare. Here is the meaning of each of them:

- **CF-Connecting-IP:** client IP address connecting to Cloudflare to the origin web server. It is supposed to be the client's real IP but in this specific case, it ends up being an IP from Cloudflare.
- **CF-EW-Via:** simply used for loop detection.
- **X-Forwarded-For:** maintains proxy server and original visitor IP addresses. In our case, the value is the same as CF-Connecting-IP. This happens when the initial request sent through Cloudflare did not contain an X-Forwarded-For field.
- **X-Forwarded-Proto:** protocol used to connect to Cloudflare. "http" or "https". With this, we could double-check that the bots using HTTPS vs HTTP matched the pattern of code 308's we were getting in the logs.
- **Cdn-Loop:** specifies how many times a request can enter Cloudflare's network before it is blocked as a looping request. Always on 1 here.
- **Cf-Ray:** hashed value that encodes information about the visitor's request and ends with the ID of the datacenter. Various datacenters were spotted in the logs. In majority the ones from Stockholm, Frankfurt and London.
- **CF-Visitor:** matches the value of X-Forwarded-Proto.
- **CF-Worker:** identifies the host that sent the request. Normally optional but it is automatically added for requests via *fetch()*, which confirms this is the command that is being abused. Cloudflare adds this header to make recipients able to filter traffic. This is good but, analysing the logs, we realize each different CFW domain only appears once. Each request was made by a different one, so blocking them via this header would not be effective.

Listing 3.9: Pseudocode to filter out obvious bots.

```
def display_obvious_bot_count():  
    botcount = 0  
    for line in log:  
        if "Cf-Worker" in line:  
            botcount += 1  
        else:  
            if user_agent is empty:  
                botcount += 1  
                continue  
            for s in suspicious_tokens:  
                if s in user_agent:  
                    botcount += 1  
                    break
```

We then added to our python script a filter for these headers and observed that 42.47% of the requests were from malicious vulnerability scanning CFWs. The advantage for us was that detecting them was easy because the CFW header fields are automatically added by Cloudflare and the malicious user behind the bots cannot remove them.

Chrome Security Fields

Paper [76] talked about the possible mismatches in the header fields arising from badly done spoofing. While we could research the entirety of the OS and browser versions and check which one is impossible to be used together, we have found something else that is interesting. A significant amount of requests had some fields starting with "Sec". Looking it up, we found they were fields added by Chromium-based browsers called User-Agent Client Hints [9]. Their goal is to facilitate content negotiation, so web developers can serve different types of content based on the client's setup. The information they contain should mirror the UA if the UA has not been spoofed, albeit in a more structured manner.

- Sec-CH-UA: list of browser brands and their version.
- Sec-CH-UA-Mobile: indicates if it is a mobile device (?1) or not (?0).
- Sec-CH-UA-Platform: OS.
- Sec-CH-UA-Arch: architecture of the device.
- Sec-CH-UA-Model: device model.
- Sec-CH-UA-Bitness: size in bits of a memory address.

Sec-CH-UA, Sec-CH-UA-Mobile and Sec-CH-UA-Platform have information that can also be found in the UA. Our hypothesis was that we could check to see if they

Listing 3.10: Pseudocode to find mismatches between Sec and UA.

```

def sec_ch_consistency():
    # Only need SEC headers if the browser is chromium
    is_chromium_ua = any(k in ua for k in list_of_chromium_browsers)
    if not is_chromium_ua and not has_sec_ua: return "SEC_NA"
    if not is_chromium_ua and has_sec_ua: return "SEC_EXTRA"
    if is_chromium_ua and not has_sec_ua: return "SEC_MISSING"

    # Platform consistency check
    for platform in possible_platforms:
        if platform in ua and platform not in sec_platform:
            mismatch = True

    # Mobile consistency check
    if is_mobile_sec_ua != is_mobile_ua:
        mismatch = True

    # Chrome major version consistency check
    if ua_chrome_major != sec_chrome_major:
        mismatch = True

    return "SEC_MISMATCH" if mismatch else "SEC_MATCH"

```

matched. In the opposite case, it would mean that the UA had been tampered with. Despite being automatically added by Chromium, we have found during our research a browser extension, ModHeader [12], that is able to tamper with the Sec fields. However, we have found no research on whether this is something commonly spoofed by bots.

Indicators of spoofing are mismatches between either the brand name or the main version number. After running the code on the logs, we found no request with mismatches between the Sec and UA information. While any human has the possibility to change their UA, granted that they have the technical knowledge, this portion of the population should be insignificant and we believe moving forward with the assumption that any spoofing is an indicator of bot, or simply suspicious activity is fair and will avoid over-complicating the training of our ML model. A forum targeted at online privacy enthusiasts would have an audience more likely to fake their UA but this is not our case here.

3.4 Feature Extraction

After the data had been collected and we knew what it was made of, we could pick the features from the data on which the model would be trained. If this step was done poorly, it would confuse the model and lead it to wrong conclusions. Back in our review of the state of the art, we have listed a few features that were used in other researches. We could start with those. However, previous research was analysing

traffic data across a whole session and take decisions based on the many requests in it. Since we aimed at taking a decision after a single request, there were many features, such as the interval of time between requests or a change of UA, that we could not use.

For our initial tests, we settled with 20 features, that we will detail in the subsequent subsections. 5 taken from the literature: the requested URI, the method used, the protocol used, the status code and the response size. They were chosen because all of them are easily accessible request fields and relevant for a single request decision. Then, we wanted to use what we learned during our traffic data analysis and add 2 features based on significant fields we had uncovered: the presence of CFW fields and the Sec fields. Lastly, we saw both in our literature review and our own analysis that the UA contained a lot of decisive information so we would definitely need to use it as well.

In Machine Learning, everything is computed using integers, which means our data fields would need to be converted to integers. For some fields such as the response size, there was nothing to do. For boolean data like the presence of a specific field, it was self-evident to interpret False as 0 and True as 1. But much of our data was of type String. To solve this issue, we used a OneHotEncoder (OHE).



Definitions

One Hot Encoding: method for converting categorical variables into a binary format by creating columns for each category where 1 means the category is present and 0 means it is not [16].

3.4.1 URI and Status Code

To make features out of the requested URI and status code, the naive approach would be to simply have a OneHotEncoder that would create a new category per URI or code in the dataset. The problem with this is twofold:

First, the resulting amount of features would end up being astronomical. Even if our website had few pages, vulnerability scanners request countless different URIs. Second, the dataset was built using our honeypot. Consequently, many URIs present in the dataset would only be found in our honeypot, not in real situations. Straight up using the URIs would result in a model that can only make decisions for a single website and not even be able to scale up with the website.

To solve those problems, we decided to group together both feature ideas. The URI plus the status code. Making a single features out of both data fields would also improve performances, as the training time of the model increases with the amount of features. Grouping features is therefore a good idea if the situation calls for it.

Considering our feature must be website-agnostic, the actual name of the accessed page is no important. What is is whether the URI is part of the website or not, like in the case of a vulnerability scan. Only valid pages will return a code 200 while

Listing 3.11: Code to filter out obvious bots.

```
# URIs of special interest
STATIC_URIS = { "/", "/favicon.ico", "/css/style.css",
               "/js/menu.js", "/robots.txt", }

def map_uri_feature(uri, status):
    if len(uri) >= 200: return "TOO_LONG_URI"
    elif uri in STATIC_URIS: return f"URI_{uri}"
    elif status == 404: return "BAD_PAGE"
    elif status == 200: return "VALID_PAGE"
    elif status == 308: return "STATUS_308"
    return f"STATUS_{status}"
```

invalid pages will return a 404 error. Our log analysis revealed code 308 was also a common occurrence for some types of bots so we also added a OHE category for that. However, the status code was not the only relevant thing, we also wanted to discriminate whether a request was made to a valid page with nothing special to it, or a page with a special significance, especially robots.txt. Robots.txt being a valid page, it would return a code 202. But to us, it was not just any other page, it was highly correlated with bot activity. At last, we added a case that would create a new OHE category with the response code in cases where it was neither 202, 404 or 308.

The code is written in Listing 3.4.1.

3.4.2 Sec Fields

There are multiple cases to take into account for Sec fields.

1. The requester has a chromium browser and generates Sec Fields matching the informations in the UA.
2. The requester has a chromium browser and generates Sec Fields that do not match the informations in the UA.
3. The requester has a chromium browser, according to the UA, but does not generate Sec Fields.
4. The requester does not have a chromium browser, according to the UA, and does not generate Sec Fields.
5. The requester does not have a chromium browser but still generates Sec Fields.

We turned those 5 possible cases into 5 categories of a new OHE. Respectively, *SEC_MATCH*, *SEC_MISMATCH*, *SEC_MISSING*, *SEC_NA* (for "Non-Applicable"), *SEC_EXTRA*. Out of these cases, only *SEC_MATCH* and *SEC_NA* were cases that could normally appear in non-modified human requests. The other ones were evidence of spoofing. The code checking these cases is described in Listing 3.10.

3.4.3 User Agent

As we saw during our log analysis, despite how easy it is to spoof the UA, a great amount of bots do not do it. From this we conclude that even though the UA is not reliable enough to make a decision based solely on it at all times, it still contains very useful information in many cases.

A problem arises when trying to make a feature out of it. Just like the URI, naively putting each new UA string as a new OHE category would result in a single feature with thousands of different possible values and would fail to match with UAs it has not seen yet. Paper [47] builds features to check if the UA is valid, invalid, empty or well-known but most other ML-based systems in the literature do not use the UA as a feature as it needs to be transformed in some way to be practical. We must then design our features in a way that solves these issues.

Our proposed method is to divide the UA feature into 2 main features and an additional 12 minor ones. First, we create a feature named UA Token Count that will simply register the number of tokens inside the UA. Simple, in theory. Because in practice we hit another annoyance. The UA is a string of characters without any formatting standard. Not only do we need to define what is actually a token, but we need to be able to extract them. For the purpose of this explanation, here is an average UA :

"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/33.0.1750.152 Safari/537.36"

While many units are formatted with the name of the technology in PascalCase followed by a slash and the version number, some feature more details in parenthesis with no particular formatting convention. Some are divided with semicolons, most use spaces both in their names and as delimiters, others use colons,... In addition, some crawlers have links to their user or company website and sometimes even include a text message.

We decided to split the UA string using the delimiters ' ' ',' '/' '(' ')' and ';' and consider the text in-between as different tokens. It allowed us to keep the important information. Normal human users will be in a similar token count range. On the other side, some bots have single-token UAs like "python" or even a completely empty UA if it has been erased. A UA can also become an anomaly long if it contains a message from the bot developer to explain the reason of the crawl and/or asking to check their website.

Next, we wanted to know whether the UA contained tokens from common browsers or tokens that were generally present. In particular, the token 'Mozilla/5.0' which should be present in all UAs. Explaining the reason for that requires a lengthy historical tangent that we shall not indulge in. But, as a summary, we put below the conclusion paragraph of the History of the Browser User Agent String [33].

Listing 3.12: Pseudocode to filter out obvious bots.

```

if ua is emty: return "NONE"

tokens.split(delimiters="[_() ;,]")
remove_empty_tokens(tokens)
remove_digit_tokens(tokens)

for token in tokens:
    # Remove version suffixes like chrome/135.0
    base = token.split(delimiters="[/]")[0]
    normal_patterns =
        ["rv:\d+", "iphone\d+",
         "sm-[a-z0-9]{5}", "[a-z]{2}-[a-z]{2}"]
    if base not in LEGIT_UA_TOKENS
        and not base.match_any_regex(normal_patterns):
            return base # bot token found

return "NON_BOT_UA" # no obvious bot token found

```

"And thus Chrome used WebKit, and pretended to be Safari, and WebKit pretended to be KHTML, and KHTML pretended to be Gecko, and all browsers pretended to be Mozilla, and Chrome called itself *Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/525.13 (KHTML, like Gecko) Chrome/0.2.149.27 Safari/525.13*, and the user agent string was a complete mess, and near useless, and everyone pretended to be everyone else, and confusion abounded." [33]

Aaron Andersen, from WebAIM - "History of the browser user-agent string"

As a last feature, we decided to create a First Unknown UA Token feature. It will go through each token until it reaches one it does not recognize as normal. Instead of making a blacklist of "suspicious" tokens we know are from bots, we made a whitelist of tokens from known browsers and OS. The type of tokens bot UAs can contain is ever-expanding. Meanwhile, the list of known browsers is much smaller. Tokens in this whitelist are not evidence of human activity, but tokens that are not in it are much more suspicious. Rather than having a feature containing a list of suspicious tokens, we stop after finding one, as it is enough to flag the UA. We believe this approach to be valid, as a same UA would not contain tokens related to different bot technologies. Googlebot would not have a GPT token. Curl will not also have an OkTTP token. So if multiple tokens in the UA happened to not be whitelisted, they would likely be related to each others. Matching these tokens required a bit more work as we only wanted to check the name of the tokens and not their version numbers or other prefixes/suffixes. The different formatting rules used by the different types of tokens made this task rather complex. We used regex patterns to run different checks and only keep the important information as shown in the code below.

3.4.4 Summary of features

At the conclusion of our feature crafting, we end up with 20 features, as follows :

- URI/Status
- Method
- Protocol
- Response Size
- CFW Field
- Sec Consistency
- UA Token Count
- First Unknown UA Token
- And 12 boolean features that match the most common normal UA fields:
ua_has_mozilla, ua_has_applewebkit, ua_has_gecko, ua_has_chrome, ua_has_firefox,
ua_has_safari, ua_has_mobile, ua_has_windows, ua_has_mac, ua_has_linux,
ua_has_android, ua_has_ios

We will refer to those features using those names in the rest of this paper.

We will further refine and adapt those features further down the line but this will be our starting point.

3.5 Implementation of the Machine Learning Model

3.5.1 Benchmark

We had mentioned the creation of a rule-based benchmark against which to compare our ML solution. The goal was to write the most rules possible that would catch bots without false positives. Essentially, it would answer the question: "How many bots can we catch without making any mistake?". Now that we had analysed the thousands of bot requests in the logs, we could have a better shot at creating this benchmark.

There are multiple ways we can easily identify bots (or malicious actors) with a high degree of confidence:

- The request comes from a CFW (presence of a CFW request header)
- The request is a vulnerability scan (the URI requested is not part of the site, but a shot in the dark to try to hit a vulnerable resource)
- The UA has been spoofed (mismatch with SEC fields)
- The UA suggests it belongs to a bot (empty/anomalous UA, clear bot token)

Running a script with these rules on the logs, 89% of the requests are considered to be guaranteed to belong to bots. In order for the results of our ML classifier to be satisfactory, it would then need to accurately catch more than 90% of bots.

3.6 Architecture

After we collected our training data and selected our features, we could implement the ML model. For that, we used the open-source Scikit Learn library [24] for Python. The Python programming language is very popular when it comes to Machine Learning and so is Scikit Learn. Scikit Learn is both feature-heavy, versatile, easy to learn and use and is actively contributed to, which makes it a prime tool for the task. Its biggest advantage is how it provides a common way to use many different models, allowing us to switch and try several ones without having to rewrite huge parts of the code. This would prove to be useful as we will see in the next subsections.

As illustrated in Figure 3.1, the system can be divided into several components. First, the classifier is pre-trained on the training data we collected. When new HTTP requests arrive on the network, they pass through a TLS Termination Proxy that allows to extract the data we need. Based on that, the classifier makes a decision. Requests labelled as human continue their normal route while those labelled as bots might be blocked or redirected.

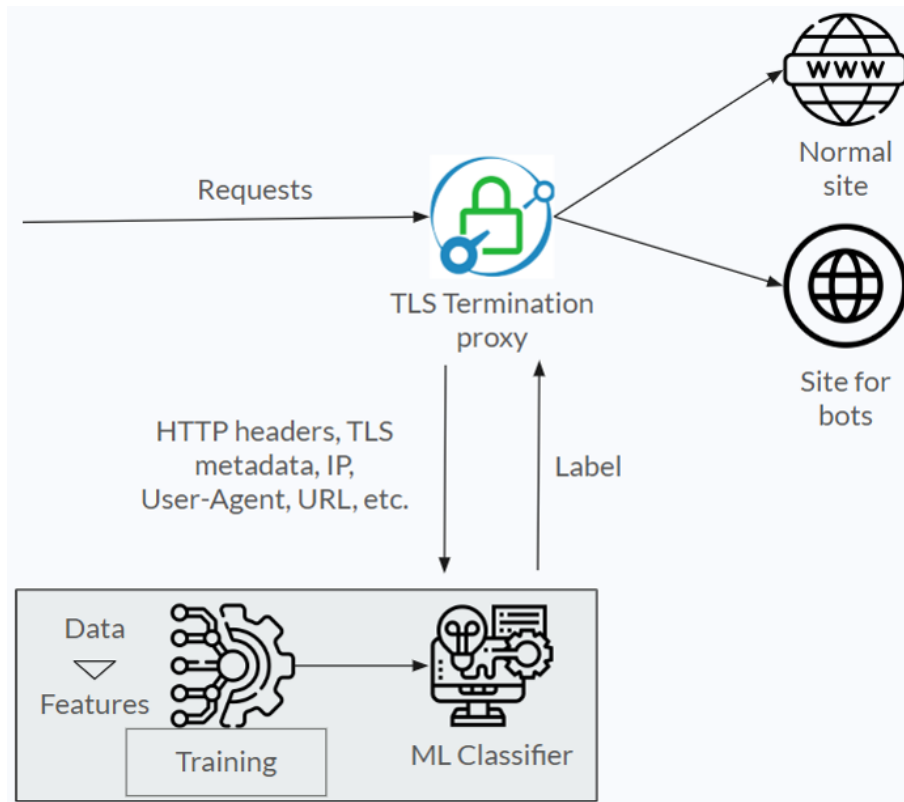


Figure 3.1: Architecture of the Machine Learning process



Definitions

TLS Termination Proxy: proxy server - an intermediary point between the clients and servers - used to terminate and/or establish TLS tunnels by decrypting and/or encrypting communications.

In our case, we use a TLS Termination Proxy to decrypt the encrypted requests and get the request fields in plaintext to pass them along to the classifier. Caddy is one such proxy. We already used it to collect data from our honeysites so using it again here would simplify the work as no more format conversion would be needed.

Considering the data from our honeypot only covered bot traffic, we only had Positive labeled data in our training dataset. Positive labeled (PL) data means it is composed of examples of the class we are trying to look for, as opposed to Negative labeled (NL) data, which is the normal class. Here, human traffic. The last possible type of data is Unlabeled data. Unlabeled data is data that we believe contains a mix of PL and NL data but we are not able or choose not to label its elements one way or the other while training the model. In our case, this would be traffic data from a normal website's logs, containing both bot and human requests. We have seen in some research papers [72] that Unlabeled data was collected and manually labelled using rules or based on Reputation Lists. In others [70], NL and PL datasets were crafted by the authors themselves. We decided to collect data from a honeypot because we believed the two aforementioned approaches would not give data that fully represents the real world because of possible assumptions during the labellisation or creation from the research teams. While the Honeypots gave us data we are sure we can label as Positive, we did not have any NL counterpart to train a model to differentiate the two classes. As a consequence, our first efforts were directed towards Semi-Supervised Learning.

3.6.1 Semi-Supervised Learning

Semi-Supervised Learning is relevant in situations where acquiring enough labeled data is prohibitively difficult or expensive, but where, on the other hand, this is not a problem for large amounts of Unlabeled data [37]. This is exactly our case as there is hardly such thing as a human honeypot, but simple traffic logs are valid Unlabeled data.



Definitions

Semi-Supervised Learning or Weak Supervision: combines supervised and unsupervised learning by using both labeled and unlabeled data to train artificial intelligence (AI) models for classification and regression tasks.

No other research we were aware of went for Semi-Supervised Learning so we researched the different possibilities of models using this paradigm in the domain of binary classifiers and by curiosity extended our search to Unsupervised Learning

which does not demand NL data. We narrowed down to three potential models: the Isolation Forest, the One-class Support Vector Machine and the Positive-Unlabeled Learning. The problem with the former two was that they were more so used for anomaly detection. In other words, it means they are made to detect a few positive outliers in a sea of NL data. Such outlier detection does not fit our use-case as around 50% of internet traffic is made of bots. Although if it keeps going in this direction, anomaly detection could end up being used, but to detect humans instead. After discarding those two, we were left with Positive-Unlabeled Learning (PUL). PUL describes classifiers which are trained on a set of PL examples and an additional unlabeled set that contains PL and NL examples in unknown proportions. It was precisely what we were after but unfortunately, Scikit Learn did not have any PUL features. Thankfully, we found a wrapper package of PUL models for Scikit Learn, named Pulearn [20], which could be effortlessly plugged into Scikit Learn.

Pulearn makes available three different PUL models: `ElkanotoPuClassifier`, `WeightedElkanotoPuClassifier` and `BaggingPuClassifier`.

The last obstacle before we were able to start implementing our solution was acquiring unlabeled data. We wanted to use the UCL's traffic logs as unlabeled training data since protecting the UCL's website was our initial situation. It was as real life data as we could get. However, making use of this data raised a privacy problem as the logs contained the real IPs of the website's users and other touchy information privacy-wise. We ended up obtaining logs stripped of any private information but it took a long time.

Implementation with Scikit Learn

The first step was to parse the log files. The PL data being captured with Caddy and the NL data being captured with Nginx, they had to be adapted to a same format. From this, the features were extracted and joined in an array *all_features*, as seen in Listing 3.14. The integer array *y* defines whether the features are positively labeled (1) or unlabeled (0). 0 is usually used as a negative label but the PUL model will interpret it as unlabeled. After that, we made a simple function *pick_model* to choose the model we want to use, in case we add more later.

Scikit Learn uses a *pipeline* object to apply a sequence of transformers on the data and conclude with a predictor. We used it in our function *build_feature_pipeline*, detailed in Listing 3.13, to process the features and add them to the model. This is done with a *ColumnTransformer*, which is a transformer that itself applies different transformers on each different column of an array. In other terms, it applies different transformers on each value depending on what feature it is part of. It is at this step that the values in our training sets are transformed into numerical values to be used for training, through the process we have described in the section on feature extraction.

Before giving the data to the model, we used the Scikit Learn function *train_test_split* to randomly split the raw data into two different sets: a training one and a test one. This is an important step because testing the model on the same dataset it was

Listing 3.13: Simplified code to build the pipeline.

```

def build_feature_pipeline(model):
    transformer = ColumnTransformer(transformers=[
        ("uri", OneHotEncoder()),
        ("method", OneHotEncoder()),
        ("protocol", OneHotEncoder()),
        ("size", SimpleImputer(strategy="median")),
        ("has_cf_worker", OrdinalEncoder()),
        ("token_count", SimpleImputer(strategy="median")),
        ("ua_tokens", OrdinalEncoder()),
        [feature_order.index(f) for f in
         ["ua_has_mozilla", "ua_has_applewebkit",
          "ua_has_gecko", "ua_has_chrome",
          "ua_has_mac", "ua_has_linux",
          "ua_has_firefox", "ua_has_ios",
          "ua_has_mobile", "ua_has_windows",
          "ua_has_safari", "ua_has_android"]]),
        ("sec_ch_consistency", OneHotEncoder()),
        ("ua_first_unknown_token", OneHotEncoder())
    ],
    remainder="drop"
    )
    pipeline = Pipeline([("preprocess", transformer),
                          ("model", model)])
    return pipeline

```

trained on might otherwise cause overfitting and mess up our evaluations. We put the parameter *test_size* on 0.2 so that the resulting test set's size would be 20% of the whole dataset. The *stratify* parameter is also used in order for each set to contain approximately the same proportion of positive and mixed data.

Finally, the last step is to call *pipeline.fit()* which is gonna train the model.

All ML models can be tuned with hyperparameters. We started with the default values from the Scikit Learn documentation and proceeded to execute a Grid Search after the first tests to find the best hyperparameters, but this will be explained further down.

Listing 3.14: Training the ML model.

```

all_logs = caddy_logs + nginx_logs

# feature extraction...
positive_features = extract_features(caddy_logs)
mixed_features = extract_features(nginx_logs)
all_features = positive_features + mixed_features
y = ([1] * len(positive_features) + [0] * len(mixed_features))
X_raw = [[row[f] for f in feature_order] for row in all_features]

X_train, X_test, y_train, y_test = train_test_split( \
    X_raw, y, test_size=0.2, stratify=y)
pipeline.fit(X_train, y_train)

```

BaggingPuClassifier is a bagging meta-estimator to learn from positive and unlabeled samples. Here we used it with a RandomForestClassifier as its base estimator. Both of them have hyperparameters.

BaggingPuClassifier:

- estimator: base estimator to fit on random subsets of the dataset.
- n_estimators: number of base estimators in the ensemble.
- n_jobs: number of jobs to run in parallel.
- max_samples: number of unlabeled samples to draw to train each base estimator.
- bootstrap: whether samples are drawn with replacement.

Random forests is a classification model that works by creating a multitude of decision trees. The output of the random forest is the class that ends up being selected by most trees.

RandomForestClassifier:

- n_estimators: number of trees in the forest.
- max_depth: maximum depth of the tree.
- min_samples_leaf: minimum number of samples required to be at a leaf node.
- max_features: number of features to consider when looking for the best split.
- class_weight: weights associated with classes.
- n_jobs: number of jobs to run in parallel.

Listing 3.15: Building the model.

```

base_estimator = RandomForestClassifier(
    n_estimators=100,max_depth=None,
    min_samples_leaf=1, n_jobs=-1
    max_features="sqrt",
    class_weight=None,)
return BaggingPuClassifier(
    estimator=base_estimator, n_estimators=10,
    n_jobs=1, max_samples=1,bootstrap=True)

```

Confusion matrix	Unlabelled	Actual bots
Predicted humans	38411	6095
Predicted bots	0	1033

Table 3.2: First results with PUL

First results

To evaluate our results, we used Scikit Learn’s features to run the model on the test set and display a confusion matrix of the results. *pipeline.predict(X_test)* is used to ask the model to determine whether each sample in the test set is positive (bot) or not. The metrics module in Scikit Learn directly has functions to get the precision and recall scores when comparing the predicted results of our classifier to the correct results *y*. We used Pyplot to display the plots and confusion matrix.

Table 3.2, the confusion matrix, shows the first results were actually far from satisfactory. All the Unlabelled were considered humans and only a fraction of bots were accurately predicted as such.

Improving the Positive-Unlabeled Learning

To make a first attempt at improving our model’s predictions, we tried to perform a Grid Search with Scikit Learn’s *GridSearchCV* class.



Definitions

Grid Search: method to exhaustively try every combination of the provided hyperparameter values by sequentially training a model on each one in order to find the best.

A *param_grid* hashmap is first built with the different hyperparameters for which we want to find the optimal value. Its keys follow the format *< model_variable > ___ < hyperparameter_name >* and it is given an array of possible values to try. We decided to start with values around the ones currently used and continue in whatever direction would show an improvement, if any.

GridSearchCV itself has parameters. In particular, its scoring parameter. We set it to *roc_auc* as this scoring measure is often used for binary classifiers with imbalanced training sets (like ours) and focuses on true/false positive rates, which

Listing 3.16: Grid Search.

```

def pu_grid_search(pipeline , X, y, model):
    param_grid = {
        # Bagging-level parameters
        "model__n_estimators": [15, 30, 50],
        "model__max_samples": [0.5, 0.7, 1.0],
        # Base estimator parameters (Random Forest)
        "model__estimator__n_estimators": [50, 100],
        "model__estimator__max_depth": [None, 10, 20],
        "model__estimator__min_samples_leaf": [1, 5, 20],
    }
    grid = GridSearchCV(estimator=pipeline, verbose=2,
                        param_grid=param_grid, n_jobs=-1,
                        scoring="roc_auc")

    grid.fit(X, y)
    return grid.best_estimator_

```

Confusion matrix	Unlabelled	Actual bots
Predicted humans	38411	6095
Predicted bots	0	1033

Table 3.3: PUL results after Grid Search

are important in our case. The Receiver Operating Characteristic (ROC) calculates the true positive rate and false positive rate at every possible threshold (= selected intervals). Graphically, a perfect model, which has always a TPR of 1.0 and a FPR of 0.0, would be represented by a line $y = 1$. The Area Under the Curve (AUC) represents the probability that the model, if given a randomly chosen positive and negative sample, will rank the positive higher than the negative. In a perfect model, it is the area under the line $y = 1$, which is the whole area of the graph. Or, numerically, 1.0.

After getting the results of the Grid Search, we modified the hyperparameters of our classifier accordingly: BaggingPuClassifier(estimator=base_estimator, n_estimators=50, n_jobs=-1, max_samples=0.7, bootstrap=True) with RandomForestClassifier(n_estimators=100, max_depth=None, min_samples_leaf=1, max_features="sqrt", class_weight=None, n_jobs = -1)

After a second run of tests, the results, in Table 3.3, were still not improving.

In the light of those results, we had to think about what could be going wrong and how to fix it. We formed several hypothesis on the issue. Maybe the fact that the data was unevenly distributed between the PL set and the unlabeled set was impacting the training. The unlabeled dataset was indeed 5 times bigger than the PL one. Or maybe we picked the wrong model or we just needed more estimators. We decreased the size of the unlabeled set and tried more combinations of models

and hyperparameters. Another problem was the lack of NL data. While PUL was meant to be trained on only PL and unlabeled data, we had nothing to check whether predictions done on the unlabeled data were even correct.

However, the actual source of the bad performances became quickly apparent to us. The two datasets came from different capturing softwares, Caddy and Nginx. Much like Caddy, Nginx is a widely used HTTP server, reverse proxy, load balancer, and more [14]. But while their logs had similar fields and overall structure, two fields needed for our features were missing in the Nginx logs: *Cf-Worker* and the *Sec* fields. Those types of headers were not captured so their feature value fell back on the default values while the caddy logs were getting actual values. This was something we were aware of but we had still decided to try training the model that way as the alternative would have just been to do away with those two features. But since all the unlabeled data was having *Cf - Worker = false* and *Secfields = SEC_NA*, that duo of features had too much weight on the model and prevented accurate predictions.

This problem demanded that we either remove those features or radically change our approach. Since NL data would be required anyway to be able to fully evaluate the accuracy of our model's predictions, we decided that we needed to acquire NL data. Our first idea was to request from a substantial amount of people to click on a specific Honeypot link, as to retrieve log data from humans. But in the end we were able to access Nginx logs from an Account-Walled part of the UCL's website, guaranteeing with a high enough probability to be fully composed of human requests. With now both PL and NL data, we decided to switch from Semi-Supervised Learning to Supervised Learning.

3.6.2 Supervised Learning

Before we could try using supervised models, we had to fix the problem with our features. *Cf-Worker* and *Sec Consistency* were using default values as the related fields were not present in the Nginx logs. It was still the case in the NL logs but the situation was different. Whereas no conclusion could be derived from the unlabeled data on the value of these missing fields, doing so was possible with our new human data. Indeed, no human request would have a *Cf-Worker* field and unless modified on purpose, *Sec* fields would be assured to be consistent too. Knowing that, we could fill the gaps in our dataset with *Cf - Worker = false* and *SecConsistency = SEC_MATCH* or *SEC_NA* based on whether the UA indicated a chromium-based browser or not.

For the models, we decided to try with several binary classifiers : K-Nearest Neighbors (KNN), Random Forest (RF), One-Class Support Vector Machine (OCSVM), Support Vector Classifier (SVC) and Log Regression. Thanks to Scikit Learn's ease of use, the structure of the code stayed mostly the same. We could just add the new models to the *pick_model* function. For the hyperparameters, we did as before and kept the default values until we did a Grid Search.

Figure 3.2, 3.3, 3.4, 3.5, 3.6 show the results of the predictions using this new

Listing 3.17: Choosing the model, supervised version.

```

def pick_model(model):
    match model:
        case "knn":
            return KNeighborsClassifier(n_neighbors=9,
                                       weights="distance", metric="minkowski",
                                       p=1, leaf_size=15)
        case "rf":
            return RandomForestClassifier(n_estimators=100,
                                       max_depth=None, min_samples_leaf=1,
                                       max_features="sqrt", class_weight=None,
                                       n_jobs=-1, random_state=42)
        case "ocsvm":
            return OneClassSVM(gamma="auto", nu=0.1, kernel="rbf")
        case "logreg":
            return LogisticRegression(max_iter=500,
                                       class_weight="balanced")
        case "svc":
            return SVC(kernel="rbf", class_weight="balanced",
                       probability=True)

```

approach. The rows of the confusion matrixes show the true labels of the samples (0 - negative - for humans and 1 - positive - for bots) and the columns show which label the classifier gave them. Which means the amount of correct predictions are shown in the top left and bottom right tiles. As it appeared, OCSVM, SVC and Log Regression were not fit for the work so we decided to move forward without them. KNN and RF on the contrary, gave incredible results. A 99% average of correct predictions.

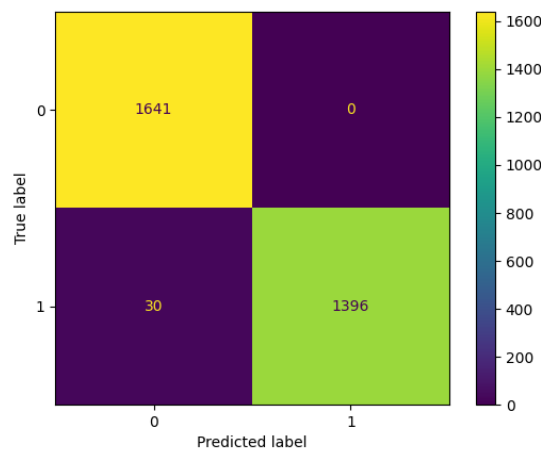


Figure 3.2: First results with KNN

After that, we performed another Grid Search, for KNN and RF this time. We tried to look for the best number of neighbors for KNN and with the different types

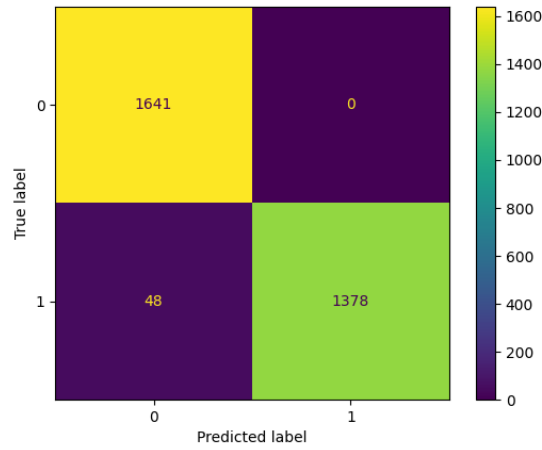


Figure 3.3: First results with RF

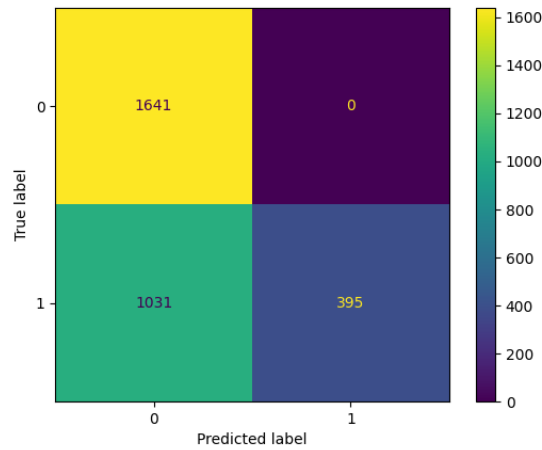


Figure 3.4: First results with LogReg

of metrics and weights. The best hyperparameters returned at the end were used and can be seen in Listing 3.18.

In addition, we calculated both the training time and the predicting time of both models.

After the Grid Search, we noted a slight increase in the average amount of correct predictions but it was minimal. We noticed that RF had more false negative errors than KNN, while both had a similar false positive rate, which was 15 to 30 times lower than the false negative rate.

Since RF was consistently slightly worse than KNN, we moved forward with KNN.

Listing 3.18: Grid Search for supervised models.

```
def pu_grid_search(pipeline , X, y, model):  
    match model:  
        case "knn":  
            param_grid = {  
                'model__n_neighbors': [1, 2, 3, 5, 7, 9],  
                'model__weights': [ 'uniform' , 'distance' ],  
                'model__metric': [ 'minkowski' , 'euclidean' ,  
                                   'manhattan' , 'chebyshev' ],  
                'model__p': [1, 2],  
                'model__leaf_size': [15, 20]  
            }  
        case "rf":  
            param_grid = {  
                'model__n_estimators': [100, 200],  
                'model__max_depth': [10, 20, None],  
                'model__max_features': [ 'auto' , 'sqrt' , 'log2' ] ,  
                'model__min_samples_split': [2, 5, 10],  
                'model__min_samples_leaf': [1, 2, 4],  
                'model__criterion': [ 'gini' , 'entropy' ]  
            }  
  
    grid = GridSearchCV(estimator=pipeline , verbose=2,  
                        param_grid=param_grid , n_jobs=-1,  
                        scoring="roc_auc")  
  
    grid.fit(X, y)
```

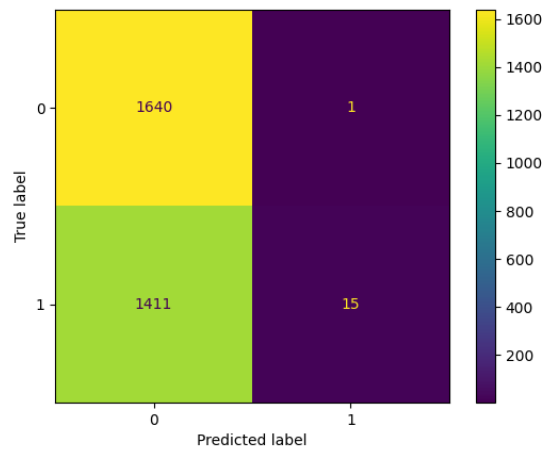


Figure 3.5: First results with OCSVM

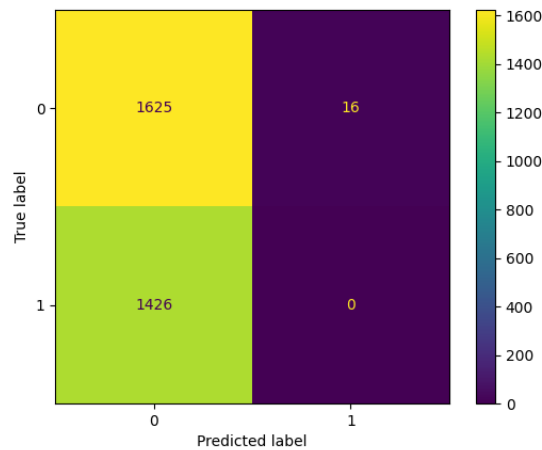


Figure 3.6: First results with SVC

3.6.3 Final Improvements

Features

Although we already had gone beyond our benchmark of beating the accuracy of our initial rule-based code, we wanted to continue looking into possible improvements. In particular, we wanted to know the importance of each feature in the decision-making of our model to better understand how it worked and whether it could be improved with a different combination of features.

KNN is a supervised machine learning algorithm generally used for classification. It works by finding the "K" closest data points - called "neighbors" - to the given input and makes a predictions based on the majority class. KNN makes no assumptions about the underlying data distribution. Hence, in addition to being distance-based, it makes it a non-parametric and instance-based learning method.

KNN being based on distance calculations leads to two things: features have no individual weights that describe the part they play in the classification, and features

Features used	False positives	False negatives
Cf-Worker + Sec Fields + First Unknown token + protocol	0%	5.05%
Cf-Worker + Sec Fields + First Unknown token + URI	0 - 0.13%	4.075%
Cf-Worker + Sec Fields + First Unknown token + protocol + UA fields + URI	0.032 - 0.065%	0.978%
Cf-Worker + Sec Fields + First Unknown token + protocol + UA fields + URI + Token count	0 - 0.016%	0.81 - 1.14%
Cf-Worker + Sec Fields + First Unknown token + protocol + UA fields + URI + Token count + Size	0 - 0.098%	0.81 - 0.978%
Cf-Worker + Sec Fields + First Unknown token + protocol + UA fields + URI + Token count + Size WITH StandardScaler	0.06%	0.39% - 0.456%

Table 3.4: Impact of features in error rates. Percentage ranges are used in case of significant variations.

whose values have larger numerical ranges of values have more impact on the results. The later can make the predictions less accurate. To fix this, we performed Standard Scaling. Standard Scaling standardizes the features by subtracting the mean, and scaling to unit variance so that the mean becomes 0 (centering the data around zero) and the standard deviation becomes 1. This transformation is applied on each feature j and sample i of the dataset's matrix. Scikit Learn facilitates this operation with its *StandardScaler* class that can be added to the pipeline inbetween the feature and model creations.

$$x'_{i,j} = \frac{x_{i,j} - \mu_j}{\sigma_j}$$

Then, to understand the impact of each feature on the model, we decided to imperically try different relevant combinations of features and compare the results. Considering that FP errors (humans classified as bots) are way more consequential than FN errors since bot users would be blocked, limited or transfered to another version of the page, our first desire was to see how many features, if any, could be used while keeping the FP rate to 0%. We found out that the combination Cf-Worker + Sec Consistency + First Unknown Token + Protocol kept the FP rate to 0% while having a FN rate around 5%. It made sense since Cf-Worker, Sec Consistency and First Unknown Token can all have a value which indicates with certainty the presence of a bot. From there, we progressively added features to see their impact of both error rates. Depending on potential future requirements, the results in Table 3.4 will help choose the best set of features to use.

The Method feature is nowhere to be found in Table 3.4. This is because our tests lead us to realize that it had no impact on the results. No matter the other features Method was used alongside to, the results were not moving. Because of this finding, we removed Method from the list of features we were using to train the

model.

The last row of the table demonstrates the benefits of StandardScaling. Thanks to it, the FN rate had been cut in two.

Despite the improvements, there is still a non-zero FP rate. Although very low, when scaled to a website with significant traffic, it could result in a non-negligible amount of normal users being flagged as bots. Depending on the chosen security measure applied to bot-labelled users, this can still cause a certain damage to the user experience. As an attempt to remediate this issue, we tested other Scalers available in Scikit Learn. We found the best results were with the MaxAbsScaler.

MaxAbsScaler works by finding the largest absolute value in the column and dividing every value in that column by it.

$$x'_{i,j} = \frac{x_{i,j}}{\max_i |x_{i,j}|}$$

The result is a data table whose largest magnitude becomes 1 and everything else ends up being scaled proportionally while zeros stay zeros. Zero preservation is very important in our case as we use sparse data.

After running a new series of tests, with the results in 3.7 and 3.8, we determined that MaxAbsScaler provides better results in terms of avoiding FP while it performs slightly worse FN-wise. Considering we had deemed minimizing FPs is more critical for our use-case, we decided to keep the MaxAbsScaler.

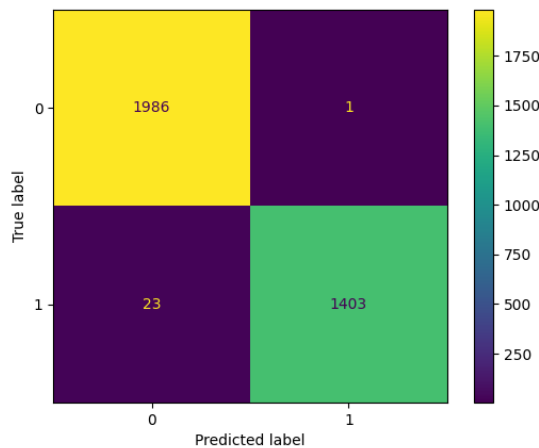


Figure 3.7: Results with MaxAbsScaler

Training data

Although our classifier was reaching more than satisfactory results, we feared that its training data - in particular, the NL human one - was too homogenous to accurately represent the broader web. As a reminder, our NL training data was extracted from logs of an account-walled part of our University's network. Making sure our results

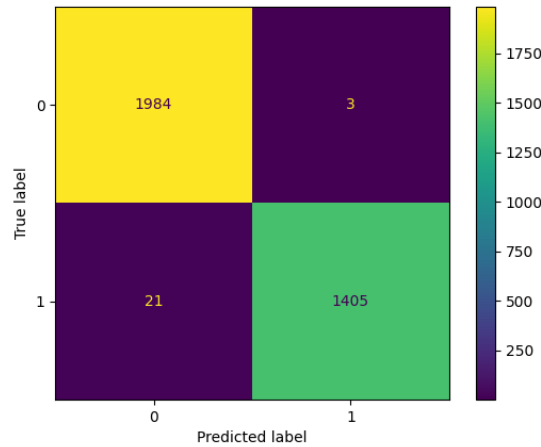


Figure 3.8: Results with Standard Scaler

could scale and adapt to any site would mean to find the means of acquiring more data that we could label as human with high enough certainty.

We went on the internet searching for any dataset that could fit, looking in different places showcasing open-source datasets, leaving no digital stone unturned. Unfortunately, no such public dataset seemed to exist. It was no surprise, the online bot issue is a very old problem. The existence of a public dataset of 100% human requests would have almost meant that this problem was solved. Either because the dataset could be used to solve it, or because the problem would have had to be solved in order to create the dataset. Meanwhile, log data from account-walled or internal networks like our University logs exist, and assembling a dataset from logs from many other institutions would solve the diversity issue. But this kind of data is most often kept private.

In the end, we decided to go for the most straightforward solution: asking people to click on a link that would log their request. This solution would not end up generating enough data to be useful for training but it could still be used in our test set to simulate more realistic and diversified testing conditions.

We started by analysing our honeypot logs and had the chance to find what we were looking for: during its months of activity, honeypot Klyth had its page *cut – scene/scene1* that had never been accessed once by a bot. This was our intuition as Klyth was the one with the robots.txt file that was forbidding bots to enter. Crawler bots from big companies were abiding by robots.txt while malicious vulnerability scanners were not, but those were not crawling the site. The result is that this page was left alone. Instead of creating a new site to log human requests, we decided to link to this specific page and we would know with high enough confidence that requests for this page during this specific time interval would be human. We mainly reached out to fellow student communities through specific channels made for research related requests. A disclaimer regarding privacy and the usage of the data was provided along with the link to ensure their informed consent. After a few

days, we collected the logs and ended up with around 150 requests.

This allowed us to collect a set of requests more diverse than the logs of the University which had a lot of requests from the same people or similar setups. Thanks to it, we could test, with better accuracy and in conditions closer to reality, the amount of FP errors our method produces.

Chapter 4

Results

In this chapter, we will look at the results of our machine learning classifier after the final improvements we made in the previous chapter. We will compare the KNN model with the Random Forest model and our rule-based benchmark. The most important metrics to look into are the accuracy of the predictions (with a bigger importance given to False Positive (FP) rather than False Negative (FN) as we want to minimize the impact on the user experience as much as possible) and the classification time. We will measure the training time as well.

4.1 Benchmark

At the start of the realization of this project, we had formulated that the goal of our ML method would be to do better than a simple rule-based method. To establish a benchmark, we ran our rule-based prototype and noted the results of its accuracy and time taken in Table 4.1. To consider our ML classifier a success, it would need to improve on the 89% accuracy of the rule-based prototype.

4.2 Accuracy

We touched on the results of the accuracy of the binary classifier in the previous chapter, but only for the intermediary steps in the creation of our model. The subsequent graphs represent the results of the final version of our method, after applying the final improvements regarding the features and the training data.

Figure 4.1 represent the amount of incorrect predictions made on test sets created from random requests from the full dataset using `train_test_split()`. The content of the test sets is random, contains 3400+ requests, different at each run and without overlap with the content of the training sets. We ran a series of these tests to be able to notice variations in the results. Results in Figures 4.1, 4.2 and 4.3. For more details, we get the average FP and NP rates for both KNN and RF in Table 4.2, as well as their variance, using the values of our test series.

As we have been able to observe previously, KNN shows better results across the board with a 99.41% accuracy.

We can further separate this 0.0123% error rate into a 0.00163% FP rate and

% of bots found	Time
89%	0.00348s

Table 4.1: Results with hardcoded rules

0.01075% FN rate. Compared to this, RF has a 0.00038% FP rate and a 0.03263% FN rate. If RF has about 3 times more FNs than KNN, it balances by having almost no FPs compared to KNN.

After that, we calculate the Precision, Recall and F1-Score. Precision is the fraction of correctly labelled instances among the instances with labelled that way. Recall is the fraction of correctly labelled instances that were able to be labelled as such. They are calculated using their respective formula:

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

The F1-Score is the harmonic mean of the precision and recall, which means it represents both precision and recall in one metric. It is a common way to combine both metrics.

$$F1 - Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

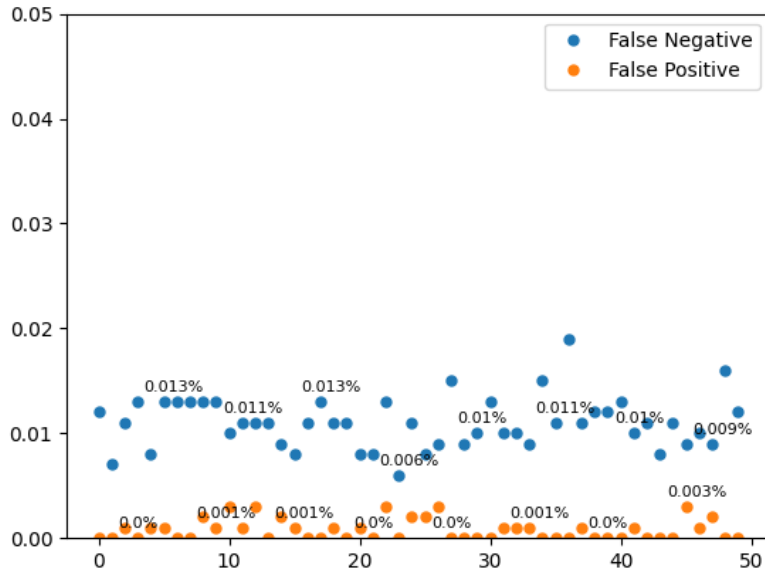


Figure 4.1: Error rates on 50 tests with KNN

4.2.1 More realistic set

As previously discussed, obtaining 100% guaranteed human requests proved to be quiet the difficult endeavor. We created a test set with the ~150 requests we had been able to obtain near the end of our project. We ran our two models against this new test set made exclusively of human requests with greater diversity than the

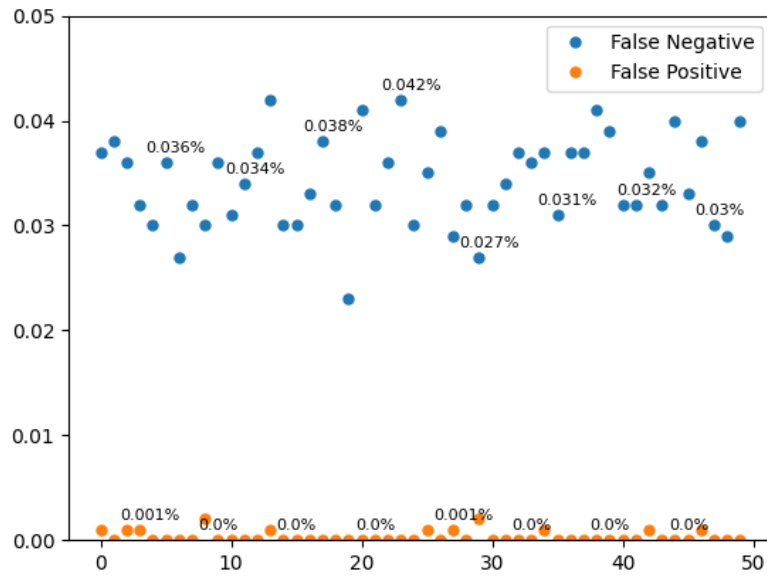


Figure 4.2: Error rates on 50 tests with RF

	KNN	RF
Average False Positive	0.00163%	0.00038%
FP Variance	0.000001	0.000032
Average False Negative	0.01075%	0.03263%
FN Variance	0.0000023	0.00000027
Total Accuracy	99.41%	98.18%
Precision	99.9%	100%
Recall	98.4%	95.6%
F1-Score	99.1%	97.7%

Table 4.2: Average results for both methods

training human data. Results about the False Positive rates are reflected in Table 4.3.

Comparing with the FP rate of the previous tests, we cannot help but notice that the error rate against the diverse human data has exploded and is about 2000 times higher for KNN. Meanwhile, RF - which previously had lower FP error rates than KNN - is completely lost and reaches 46% of errors, which is not far from a random predictor. In addition, RF has a very high variance and its results vary a lot from one test to another.

Alternative models

By looking at the samples causing errors, we can try identifying the causes of these misclassifications. To benefit from a greater pool of samples, we look at the errors coming from RF. We notice that the request header data forming misclassified samples are very close to the data of well classified ones. Oftentimes, all fields but

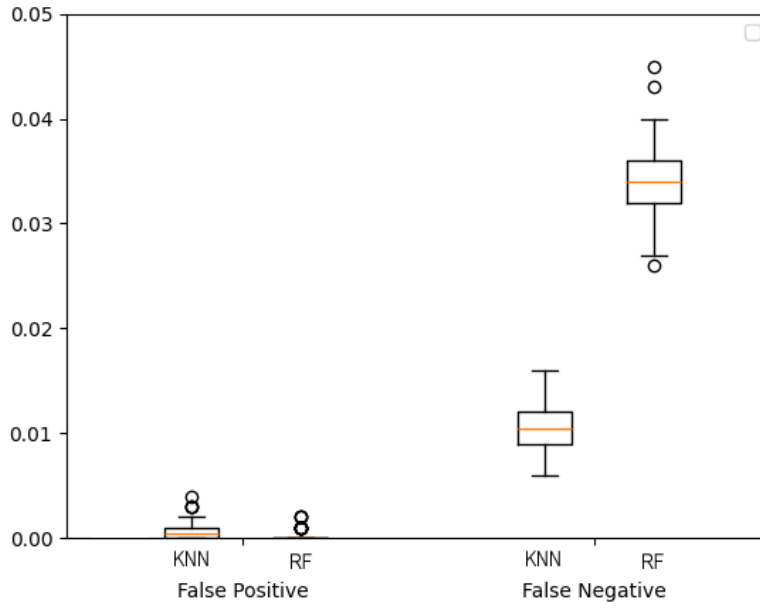


Figure 4.3: Error rates on 50 tests: summary

	Average FP rate (diverse data)	Variance	Average FP (previous tests)
KNN	3.2%	0	0.00163%
RF	46%	4.5	0.00038%

Table 4.3: False Positive rates for test set with diverse human data accross 50 tests

one have the same values. We therefore form the hypothesis that removing the features linked to these fields might remove the cause of error. By analysing all error-causing samples, we identify the problematic features. The problem is due to the homogeneity of the training data, which leads the model to take incorrect decisions when encountering underrepresented values. In order of consequence on the error rate, below is the subset of features leading to such issues, along with our hypothesis as to the reason why.

- **Protocol:** the negative-labelled training logs only had HTTP/1.1 as Protocol so encountering anything else causes the sample to be classified as a bot.
- **Response size:** we believe that the fact that the training and testing samples are taken from 2 different sites plays a role here, as response sizes would on average be different.
- **UA data on OS/browser/device** (e.g. has_chrome, has_apple,...): probably caused by the homogeneity of the training set. While the dataset contained a great amount of requests, the amount of different people and setups represented was rather low. Requests coming from people with vastly different setups were therefore flagged as bot.

- **UA Token count:** causes very few errors, likely for the same reason as above.

In the light of those discoveries, we make the attempt to train new models, KNN-A / RF-A, this time without using the features negatively affected by the homogeneity of the human training set. Doing so makes the error rate drop. By taking a look at the remaining samples misflagged as bot, we notice that they actually really look like bots. Indeed, they all are sent by an IP belonging to Google and have the following UA:

```
"Mozilla/5.0 (Linux; Android 10; K) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/138.0.0.0 Mobile Safari/537.36 (compatible; Google-Read-Aloud;
+https://support.google.com/webmasters/answer/1061943)"
```

What happened? Did a bot slid into our carefully gathered human data? Not really. It is actually an edge case we had not anticipated. Google Read Aloud is Google's text-to-speech feature. When and only when triggered by a user, it runs and reads the page's text aloud. One of the users from which we collected the data must have had it on. This illustrates well how the diverse set of existing technologies can create many edge cases. So many that it is hard to design a system that would cover it all. By default, it considers Google-Read-Aloud as a bot and the inclusion of a URL in the UA is doubly suspicious. If we want to consider Google Read Aloud as a human-driven request, the way the First Unknown Token feature is extracted should be changed. Since the default behavior is technically not a classification failure and it does not affect the classification of the actual user using the service, we decide to take the Google Read Aloud samples off the human test set.

Relevant metrics about how many bots are accurately caught and the amount of false positive are compared in Table 4.4 between the normal models with all the features and the new adapted models. As a memory refresher, the features used in each model are displayed in Table 4.5.

We observe that once the problematic features stop being used, the false positive rate drops significantly, albeit accompanied with a small drop in detection accuracy, which was to be expected.

The next logical step is to try testing a model trained on this diverse data. We use once again Scikit Learn's `train_test_split()` to split the dataset into a training and test sets. Our first test is replacing the original human dataset by the more diverse one (KNN-B). In a second time, we test merging them both together (KNN-C). Looking at the results in the last columns of Table 4.4, we find that the results significantly improve. Despite using all the features, we are able to achieve very low false positive rates.

4.2.2 Server-agnostic model

Some features of our models are based on data some servers, like Nginx, cannot access out of the box. In Table 4.6 we test the accuracy of the bot detection using

	Full KNN	Full RF	KNN-A	RF-A	KNN-B	RF-B	KNN-C	RF-C
Accuracy	99.41%	98.18%	93.24%	92.94%	98%	96.19%	98.52%	96.63%
Recall	98.4%	95.6%	84%	83%	98%	96.1%	96.7%	92.1%
FP rate on diverse test set	3.2%	46%	0%	0%	1.72%	1.72%	0.22%	0.149%

Table 4.4: Comparison of the models with all the features against the ones adapted to minimize the side effects of poor data diversity.

Models	Server agnostic features	Caddy-specific features
Full KNN	URI/Status, Protocol, Response Size,	
KNN-B	UA Token Count, Common UA tokens,	CFW Field, Sec Consistency
KNN-C	First Unknown UA Token	
KNN-A	URI/Status, First Unknown UA Token, has_mozilla	CFW Field, Sec Consistency

Table 4.5: Feature comparison of the models

only the set of features that can be captured by all servers. Again, we compare our original model with the one adapted to take into account the lack of diversity in the human training data, which uses less features, and the one trained on the more diverse data.

Without surprise, we observe a severe drop in recall for the KNN/RF-A models, having only the URI, Status, has_mozilla and Token Count features left. This is not enough to take correct decisions. The other models still have most of their features enabled so they do not encounter the same problem and the decrease in accuracy and recall is almost insignificant. Strangely, Full RF and RF-C seem to have slightly better recall without the Caddy-specific features.

4.3 Time

The second important metric to take into account is the time the classifier takes to make a decision on whether the incoming request comes from a human or a bot. an ML-based method has two steps: the training and the classification. The model's training time is not very important as it does not have to be done in real time while the request are coming. Training the model can be done before it is deployed.

	Accuracy with Caddy features	Accuracy without Caddy features	Recall with Caddy features	Recall without Caddy features
Full KNN	99.41%	99.3%	98.4%	98.4%
Full RF	98.18%	98.18%	95.6%	96.1%
KNN-A	95.3%	94.38%	84%	49.5%
RF-A	94.5%	81.62%	83%	48.5%
KNN-C	98.52%	98.5%	96.7%	96.7%
RF-C	96.63%	96.94%	92.1%	92.9%

Table 4.6: Tests without features enabled by Caddy-specific headers.

Model	Average training time	Average prediction time	Variance (prediction)
KNN	0.1s	0.0015s	0.0000004
RF	0.38s	0.0013s	0.0000002

Table 4.7: Time statistics for both models

Table 4.7 shows the average training time of both models in their final form.

Classification time has much higher stakes. If the user has to wait for the model to take its decision in order to access the page, times of significant length will negatively impact the user experience, like Captchas do. On the other hand, if the user is allowed to browse normally until the decision is made, a lengthy decision time could reduce the usefulness of the whole process since we have seen in the state of the art that bots go in and out the websites very quickly (less than 3s) and make very few requests (in many cases, only one).

Figure 4.4 and 4.5 reproduce the time taken to classify the series of 3000+ requests in the tests of the previous section. As included in Table 4.7 RF averages at 0.0013s per batch, while it is 0.0015s for KNN. In conclusion, KNN has a lower training time but RF performs better than KNN when it comes to prediction time and its variance is lower too.

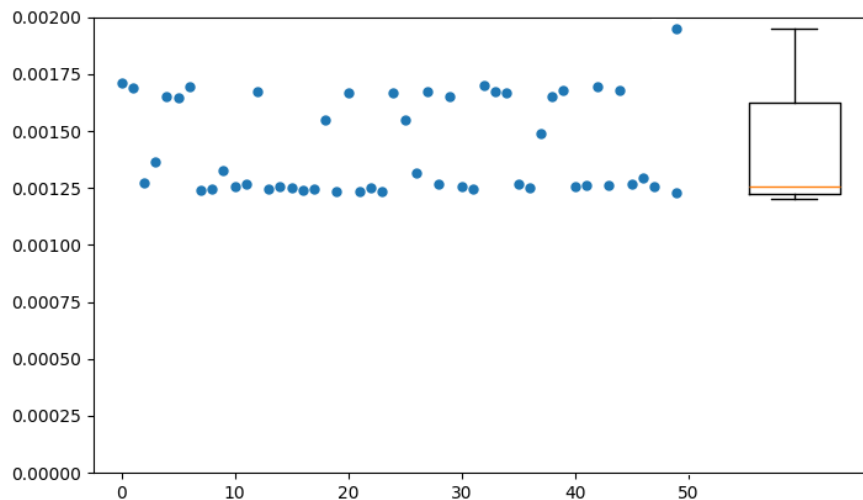


Figure 4.4: 50 tests of the time (seconds) taken to make 3400 classifications with KNN

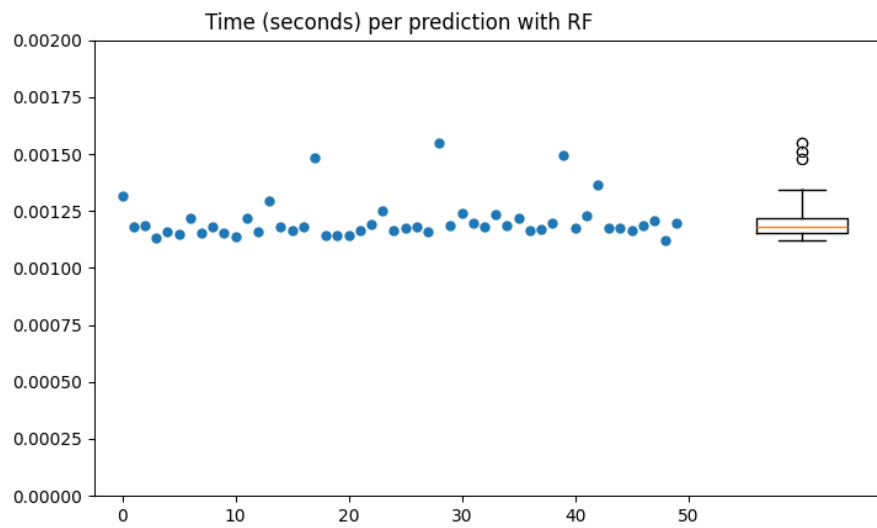


Figure 4.5: 50 tests of the time (seconds) taken to make 3400 classifications with RF

Chapter 5

Discussion

In this chapter, we will analyse and talk about our method's results presented in the previous chapter.

5.1 False positive on diverse human dataset

We have seen that the model made way more false positive errors on the dataset composed of more diverse human data than the logs it was trained on. But we were able to bring the error rate down by crafting a new model short of some features.

What it means is that some of the features were greatly affected by the lack of diversity from the human training data in their associated header fields. When encountering a data point that was underrepresented or absent in the training data, it was flagged as bot-like. For example, in the training data, bots use varied protocols but humans only use HTTP/1.1. The model was therefore 100% convinced that any value other than HTTP/1.1 could only be the sign of a bot. This gave Protocol a huge weight in that decision. The same thing goes for the other features affected by the lack of diversity. Once those features were removed, the false positive rate goes back to a value comparable to our results when testing on a subset of the logs from our University (Table 4.2). From that, we hypothesize that if we were to obtain a large and diverse enough human training set, we could reach the same results using the normal model with all the features, as no feature would be affected by a lack of data diversity anymore.

While we cannot confirm this hypothesis with 100% certainty, our next experiment (KNN/RF-B and KNN/RF-C) made us confident that this would be the case. Using the diverse human data to train a classifier gave us results with low false positive rates despite using all the features.

But even if its addition makes our training set more diverse, this additional dataset cannot fully claim to represent the wider web because of its small size. To be ensure the efficiency of our method at scale, larger human datasets (with data diversity) would be needed.

5.2 Comparison with state of the art/related works

We will start by comparing our results to our benchmark and the related works in the field. Table 5.1 summarizes this comparison. We use our original KNN and RF models, which represent the best results we were able to reach with this method (but result in more errors on the more diverse test set), and KNN-C / RF-C, which gave the best results on the more realistic conditions and is therefore the most usable in practice at the present moment.

	KNN	RF	KNN-C	RF-C	Benchmark (rule-based)	Neural Network [72]	C4.5 decision tree [73]
Accuracy	99.41%	98.18%	98.52%	96.63%	89%	99.31%	90%
Precision	99.9%	100%	99.7%	99.7%	99.8%	98%	82%
Recall	98.4%	95.6%	96.7%	92.1%	89%	95%	95%
F1	99.1%	97.7%	98.3%	96.4%	98.4%	96%	88.023%
Time	0.0015s	0.0013s	0.0015s	0.0013s	0.00348s	/	/
Requests needed	1	1	1	1	1	6	4
Data Gathering	Honeypots	+ private	logs		Honeypots	Heuristics + Reputa- tion list	Heuristics, mainly UA-based

Table 5.1: Comparison of different methods.

When it comes to accuracy, our method manages to surpass the benchmark, and perform consistently better than one of the state of the art ML methods, namely, the C4.5 decision tree [73]. Meanwhile, KNN-C comes just short of outperforming the Neural Network from [72]. It has slightly better recall but slightly worse precision, which means it makes slightly more false positive than [72]. Let us remember that the biggest difference between our method and the others is that we do real-time classification on a single request while the rest of the research focuses on session-based classification. So whereas some methods will take a decision after 4 or 6 requests, our method only does it in 1.

5.2.1 Why those results?

Accuracy

Filtering bots from humans is a notoriously difficult problem, so why are we able to reach such low False Negative error rates? As seen in the results section, our models have been able to perform better than the benchmark made of hardcoded rules.

We believe the primary reason is the bot landscape and the features we used. As we discovered during our log analysis, the overwhelming majority of bots captured by both our honeypots were vulnerability scanners. More precisely, maliciously repurposed Cloudflare Workers. And Cloudflare Workers are extremely easy to filter as they have multiple request header fields linked to their Cloudflare Worker nature. At the time of our research, no other research paper talked about this topic, even the most recent ones. Hence, such data has never been used as a feature to filter out bots before. This can explain our low amount of False Negatives.

Data gathering

Additionally, but unfortunately, the higher rate of False Positives on the main model can easily be explained by the lack of diversity in our human (negative-labelled) training set. We chose to only use in our training set requests that we could be 100%

sure to come from humans because our analysis of the state of the art taught us that no rule-based or heuristic technique exist to accurately filter out bots with the most advanced camouflage from humans. As seen in Table 5.1, once we test on more realistic conditions (more diverse test set) our method (KNN-C / RF-C) positions itself between two ML-based solutions from the state of the art. But the way they gather data for their datasets is not bulletproof. While some heuristics are effective, one mostly use the UA as a criteria to classify requests as bot or human [73] and the other relies greatly on the Udger reputation list [72]. The former does not address UA spoofing, which is a common issue and the main reason why the UA alone is unreliable. The later claims to only keep in their dataset the requests whose nature they were 100% sure of and discard the others. But in both cases, it creates a dataset that is not representative of the broader web. The models only end up being trained and tested on data from the simpler bots and do not address bots that are efficient at camouflaging. Which is the biggest and rising problem and the main reason why we would want to take advantage of Machine Learning’s capabilities. Just like our original models, it is possible that given more diverse data to test on, or put in a real situation, those state of the art models would also perform less well.

In the end, building a bot dataset and a human dataset from a mixed dataset was impossible without making a compromise on accuracy and poisoning our training data to some extent. Since unmasking the best hidden bots was the point of using ML over hardcoded rules in the first place, we believe compromising on the quality of the training data for this specific kind of bot would have defeated the purpose of the project.

The unfortunate side-effect was that it lead to a lack of diversity in the human data. To counter this, we first opted for trying Positive-Unlabelled Semi-Supervised Learning, but the results were suboptimal. When we switched to Supervised Learning, we had to come face to face with the fact that it was highly difficult to obtain a large dataset of 100% human data. Still, using logs from the University allowed us to create a working prototype for which we could verify the accuracy of the predictions and we were able to establish its efficacy at getting low FN. The more diverse but smaller set of human request we were able to get near the end of the project showed us that our model had a higher error rate on these samples. Which was expected, because of their difference to the training data. If it were to be deployed on a website, it could cause some issues though, as even a low percent of errors, scaled to an entire website’s traffic, would lead to problems. To remediate, we created KNN-A and RF-A to classify samples without using features impacted by the training data’s homogeneity. Then we backtracked and integrated the diverse human data into our training set. KNN-C / RF-C proved that this better training data lowers the error rate. This showed our method does constitute a good proof of concept and we are confident that given more diverse human data to train on, the results would reach the same heights as before.

5.3 Limitations

5.3.1 Efficiency at scale

If our method can filter out bots from humans with more than 90% accuracy, does it mean the problem of bot on the internet is mostly solved? No. The increase of the online bot presence brings two major issues: the overloading of web servers and the theft of data by web scrapers. The original goal of this project was to create a detection method able to detect bots to further take actions on that would reduce the server load. This is what this project has focused on.

But the second major problem with bots demands a whole other approach. Web scrapers, price trackers, and other targeted attacks are the doing of much more determined bot developers. The bots engineered for such tasks are way better at camouflaging as a real human and modify their behavior in real-time to bypass or avoid triggering detection methods. Often they are created specifically for a certain task on a certain website. Our method performs better than rule-based ones, which implies the ML model can catch on patterns non-obvious to the human eye. But it has not been designed to catch on specific advanced bot behavior. Other than being out of scope, the reason is simple: we obtained our training data from honeypots. These honeypots would not be the victims of targeted attacks therefore no data from these advanced bots ended up in our dataset.

In conclusion, a website wishing to reduce the load on its servers might find our method useful to detect bots, but a journalism website targeted by competitors' web scrapers would not be able to efficiently protect their data with our method.

A question immediately follows: if we were to train our model on more advanced bot data, would it perform better? Considering the abundance of ways bots can behave to fly under the radar, it would be difficult to predict if all of them can be detected using the same method, even if enough data would be at our disposal. Of course, the ever-changing nature of the bot landscape means we would need to constantly update our training data with data on the new behaviors and counters. The advantage of ML is that doing so is a possibility, which reduces the need to constantly build new protections on top of each others. But updating the model being possible does not mean that it is easy. The biggest obstacle is the very nature of targeted attacks. To best protect against them, training data should be captured from the victim website's own traffic data. A general purpose model could not know the intricacies of every use-case. Unless this general purpose model would get its data from every targeted website. However, websites prefer not to share such data. Consequently, they end up having to build their own defenses. Of course, there is a sort of exception to that: Cloudflare. Cloudflare's CDN services are widely used and it makes them able to monitor and study massive amount of traffic data from all kinds of big websites. It made them able to add a robust - albeit not perfect either - bot security to their range of services. Without getting data from another service, like Cloudflare does, it is hard to believe any ML-based solution could perform as well across the board.

5.3.2 Evolutions of technology

To stay on the topic of Cloudflare, it is important to note that the evolution of the bot landscape might not only require the model to be fed with new data, but also to update the very features it uses for its classification. As discussed, a security flaw with Cloudflare Workers allowed them to be hijacked to act as vulnerability scanners. While this flaw is not new and has not been fixed at the time of writing, it could be in the future. This would create the need to reconsider the use of the Cloudflare Worker feature we use in our model. The same could happen to other features.

In fact, bots are not the only ones to evolve, humans will too. Not so much with their behavior but with the technology they use. For example, with the feature that detects anomalous tokens in the User Agent. When new browsers or hardware eventually come out, their UA token will have never been encountered in the training set so they will get flagged.

Another possibility, although less likely, would be a change in the request structure, like a header that becomes obsolete and goes unused. Or more positively, new data points in the request that would make it easier to detect bots. In that case, the model would benefit in supporting them as features.

One last thing to take into account is the unknown unknown. When gathering human data closer to real-life conditions, we faced an edge case we had not anticipated: Google Read Aloud. A tool emitting requests that look exactly like a bot, but is in fact a user-triggered feature. While we could adapt our feature extraction algorithm to flag this UA token as bot or human depending on the needs of the user, there is no telling what other edge case we do not know about. Technology is diverse and other tools or requests could live on the edge of being considered bot or human. Either we should seek to adapt our method to each new edge case being found or we should content ourselves with the default behavior. This is the type of dilemma that each researcher in AI and ML faces at some point.

In conclusion, our ML solution is not meant to be a monolith never to be updated.

5.3.3 Present usability

In the first part of our research, we have familiarized ourselves with the many techniques that protect against the many kinds of bots. We have learned that each has its own pros and cons, its requirements and its use-cases in which it performs better. Our method is no different. Depending on context and objectives, one might prefer, or not, to use our method.

We touched on the use-cases above. For the requirements, the important thing to remember is that both the training data and the incoming requests to evaluate must be able to go through a TLS Termination Proxy placed at the network's edge. Without it, we cannot access the request header values used in the decision process. The Proxy must then of course also be configured correctly as to be able to capture the required fields. Here, we used the Caddy server. But some other servers, like Nginx, do not have the ability to log some of the headers like the Cloudflare Worker or the Sec-UA ones. This could make the full version of our method inaccessible

for users of those servers without a change of technology. However, the tests with the results presented in Table 4.6 show the decrease in accuracy due to the missing features is pretty low overall. Thus, it is still usable.

Regarding the current state of our solution, it is working but might require adjustments depending on the desired outcome. For starters, it currently classifies all bots in the same "bot" category. But the way the features are extracted allows for the user to customize the profile of bots to exclude from this filter. Tokens for a specific benign bot might be added to the whitelist for example and will then not be picked up by the feature that looks for suspicious tokens in the UA.

Furthermore, we have seen that the different result metrics can be influenced by the various parameters and building blocks of our method. Depending on needs, tweaking the method is possible to reach different objectives. It is all a matter of compromise.

For example, in Table 3.4, we investigated the impact of features on the results and found out the feature group "Cf-Worker + Sec Fields + First Unknown token + protocol" was able to reach 94.95% accuracy with minimal False Positive errors. In the case where such a FP error rate is a hard requirement, opting to do away with the rest of the features (and increasing the False Negative rate) might prove to be worth it. And by adding the diverse dataset to the human training set, we obtained the models KNN-C and RF-C, which overall performed better while keeping a low False Positive rate.

However, the main limitation currently is still the lack of diversity of human data. It lowers the reliability of human request identification and thus causes more False Positives, which is a problem in many cases. To take the method to its full potential, it should be trained on more human requests (with high diversity). Even after adding our more diverse dataset, this is still relevant because that dataset is small and while it is *more* diverse, it still cannot represent a large enough portion of the wider web.

5.4 CyberSecurity analysis of the project

When creating a project aiming at improving cybersecurity, its own security must be considered and ensured as to not end up with a project that creates more security issues than it fixes. For this reason, it is commonplace to assess the risks associated with it.

After pondering on the question, we do not believe that our project in itself can introduce vulnerabilities. However, the major risk is over-reliance. We have at multiple occasions discussed that each defense against bots has its own ups and downs and that a single one will not cover by itself the entire range of bot threats. Our project is no different. Installing it and then forgetting about it is not enough to be fully protected. Thinking otherwise would make one vulnerable to the more advanced bot threats.

Considering it is based on Machine Learning, it is however vulnerable to common attacks on ML systems. In particular, data poisoning, which would naturally lead to misclassification, which could either make the website vulnerable to specific threats or trigger the blocking of normal users.

To mitigate the risks, policies regarding data gathering should be established and followed.

In addition, like every other anti-bot technology, if the adversary were to learn or reverse engineer how it works (e.g. features), they could design their bots specifically to bypass it.

Since our method was not deployed on the University, specificities linked to implementation could not be tested. But it is likely that the system could be victim of Denial Of Service attacks. An attacker sending a large enough amount of requests at once could overload the system. If the users are required to wait the normally small classification time in order to continue browsing, overloading the system could lead to significant wait times and a degradation of the user experience, causing reputational, financial or other kind of damage depending on the website. Similar events have already happened with services, such as Cloudflare, going down. As mitigation for this, and the other risks, we recommend once again to deploy multiple different kinds of bot protections to critical infrastructure.

Chapter 6

Future work

In this chapter, we explain the important things to have in mind when continuing on improving this method or when exploring others.

The main area of improvement that would benefit from further work is the constitution of the human training set. Asking people to fall into a honeypot gave us data of good quality but our campaign was relatively small. Scaling it up would allow to better train the model on that data, instead of just using it to test the result. Of course the difficulty resides in finding a way to gather large enough amount of requests while keeping the certainty that they come from humans. With a little bit of chance, seeking partnership with other Universities, research centers or private entities could reveal to be useful.

The key is to ensure that the data remain diverse. By diverse, we mean that each possible value for each feature should be significantly represented in the dataset. In particular, UA tokens. Tokens about hardware, browsers, browser engines, all with different versions and combinations. The goal being to minimize blindspots.

Many of the key features used in our model were built from data that had not been used to create features in previous research, or built differently than in the state of the art. These features were found by analyzing traffic logs. It is possible that more hints and patterns of bot activity could be found with further analysis. Furthermore, it is not only human data that we would benefit in getting more of. With bot data from real websites with a real threat landscape, more patterns and types of bots could possibly be uncovered.

Moreover, testing the system in real conditions by deploying it on a network would be the logical next step.

Chapter 7

Conclusion

In recent years, the amount of bots crawling the web has greatly increased, with no sign of stopping, causing many issues from intellectual property theft to environmental consequences. In this paper, we took a look at the diverse ways previous researchers have found to attempt to protect, fully or in part, websites against those bots. We have established the different types of bots and the different kinds of purposes they are used for. Then, we suggested a detection mechanism based on Machine Learning with the aim of filtering bots from humans in order to reduce server load. Our approach operates on HTTP requests sent to the server. They are intercepted by a TLS Termination Proxy and a subset of this data is used by the model to classify the request as either human or bot. Our analysis of logs coming from our Honeysites allowed us to identify data points that had not been used before as model features. Specifically, the Cloudflare Worker headers and Chrome's Sec-related headers. In addition to that, we designed a new way to extract significant data from the User Agent to optimize classifier results. To catch bots in real time as fast as possible, we made our classifier operate on single requests instead of on sessions or batch of requests. The results showed promising results with our classifier performing better than hardcoded rules with a very low amount of false negatives. However, due to the lack of available human data to train on, it struggled to correctly identify humans, leading to a low, but significant at scale, false positive rate. While we made the necessary modifications to the models to correct this tendency, the only real solution is to train it on more human data. Despite this setback, our results make us optimistic in the belief that the method would achieve even better results given more data to complete its training.

Bibliography

- [1] ai.txt: A Domain-Specific Language for Guiding AI Interactions with the Internet, <https://arxiv.org/html/2505.07834v1> [Cited on page 13.]
- [2] BotScout.com: Proactive Bot Detection, Screening, & Banning, <https://botscout.com/> [Cited on page 15.]
- [3] Easy Web Extract Download, <https://easy-web-extract.apponic.com/> [Cited on page 8.]
- [4] fingerprintjs2 - npm, <https://www.npmjs.com/package/fingerprintjs2> [Cited on page 23.]
- [5] font-detect-js/flash/FontList.swf at master · gabriel/font-detect-js, <https://github.com/gabriel/font-detect-js/blob/master/flash/FontList.swf> [Cited on page 23.]
- [6] Glaze - Protecting Artists from Generative AI, <https://glaze.cs.uchicago.edu/> [Cited on page 21.]
- [7] Good Bot, Bad Bot: Characterizing Automated Browsing Activity, <https://ieeexplore.ieee.org/abstract/document/9519384> [Cited on pages 7, 9, 10, 12, 15, 16, 19, 24, and 31.]
- [8] HUMAN Security, <https://www.humansecurity.com/> [Cited on page 23.]
- [9] Improving user privacy and developer experience with User-Agent Client Hints | Privacy & Security, <https://developer.chrome.com/docs/privacy-security/user-agent-client-hints> [Cited on page 40.]
- [10] LevelBlue - Open Threat Exchange, <https://otx.alienvault.com/> [Cited on page 15.]
- [11] Modernizr: the feature detection library for HTML5/CSS3, <https://modernizr.com/> [Cited on page 23.]
- [12] ModHeader - Modify HTTP headers, <https://modheader.com/> [Cited on page 41.]
- [13] Mozenda - Scalable Web Data Extraction Software & Services, <https://www.mozenda.com/> [Cited on page 8.]
- [14] nginx, <https://nginx.org/> [Cited on page 54.]
- [15] Nightshade: Protecting Copyright, <https://nightshade.cs.uchicago.edu/whatis.html> [Cited on page 21.]

- [16] One Hot Encoding in Machine Learning, <https://www.geeksforgeeks.org/machine-learning/ml-one-hot-encoding/>, section: Machine Learning [Cited on page 42.]
- [17] Our Services | Project Honey Pot, https://www.projecthoneypot.org/services_overview.php [Cited on page 15.]
- [18] PluginDetect, <http://www.pinlady.net/PluginDetect/> [Cited on page 23.]
- [19] Protecting Creatives or Impeding Progress?, <https://openfuture.eu/blog/protecting-creatives-or-impeding-progress> [Cited on page 13.]
- [20] pulearn – Positive-unlabeled learning with Python., <https://pulearn.github.io/pulearn/> [Cited on page 49.]
- [21] Puppeteer, <https://pptr.dev/> [Cited on page 10.]
- [22] Robots d’exploration communs de Google | Infrastructure d’exploration Google | Crawling infrastructure, <https://developers.google.com/crawling/docs/crawlers-fetchers/google-common-crawlers?hl=fr> [Cited on page 9.]
- [23] samyk/evercookie: Produces persistent, respawning "super" cookies in a browser, abusing over a dozen techniques. Its goal is to identify users after they’ve removed standard cookies and other privacy data such as Flash cookies (LSOs), HTML5 storage, SilverLight storage, and others., <https://github.com/samyk/evercookie> [Cited on page 23.]
- [24] scikit-learn: machine learning in Python — scikit-learn 1.8.0 documentation, <https://scikit-learn.org/stable/> [Cited on page 47.]
- [25] Scrapy, <https://scrapy.org> [Cited on page 8.]
- [26] Selenium, <https://www.selenium.dev/> [Cited on page 10.]
- [27] simonw/shot-scraper: A command-line utility for taking automated screenshots of websites, <https://github.com/simonw/shot-scraper?tab=readme-ov-file> [Cited on page 22.]
- [28] ThreatMetrix: Automated Risk Management & Fraud Detection, <https://risk.lexisnexis.com/products/threatmetrix> [Cited on page 23.]
- [29] Threats of price scraping on e-commerce websites: attack model and its detection using neural network | Journal of Computer Virology and Hacking Techniques, <https://link.springer.com/article/10.1007/s11416-020-00368-6> [Cited on pages 7, 9, and 20.]
- [30] Track, control, and optimize your website for AI agents and bots, <https://knownagents.com/> [Cited on page 15.]
- [31] User Agents Analysis :, <https://udger.com/> [Cited on page 15.]

- [32] Web Content Extractor - The most efficient web scraper, <https://www.newprosoft.com/> [Cited on page 8.]
- [33] WebAIM: History of the browser user-agent string, <https://webaim.org/blog/user-agent-string-history/> [Cited on pages 44 and 45.]
- [34] The world's fastest framework for building websites, <https://gohugo.io/> [Cited on page 32.]
- [35] Xetera/ghost-cursor: Generate human-like mouse movements with puppeteer or on any 2D plane, <https://github.com/Xetera/ghost-cursor> [Cited on page 26.]
- [36] dnif-archive/enrich-badip (Dec 2023), <https://github.com/dnif-archive/enrich-badip>, original-date: 2018-03-18T11:37:48Z [Cited on page 15.]
- [37] What Is Semi-Supervised Learning? | IBM (Dec 2023), <https://www.ibm.com/think/topics/semi-supervised-learning> [Cited on page 48.]
- [38] AI-Driven Bots Surpass Human Traffic - Bad Bot Report 2025 (Jun 2025), <https://cpl.thalesgroup.com/about-us/newsroom/2025-imperva-bad-bot-report-ai-internet-traffic> [Cited on pages 1 and 36.]
- [39] Cloudflare HTTP headers (Oct 2025), <https://developers.cloudflare.com/fundamentals/reference/http-headers/> [Cited on page 39.]
- [40] (PDF) Web robot detection: A probabilistic reasoning approach. ResearchGate (Aug 2025) [Cited on page 25.]
- [41] Trapping misbehaving bots in an AI Labyrinth (Mar 2025), <https://blog.cloudflare.com/ai-labyrinth/> [Cited on pages 27, 28, 31, and 32.]
- [42] URGENT: Malicious requests coming from Cloudflare Workers (2a06:98c0:3600::103) - Application Security / SSL / TLS (Jun 2025), <https://community.cloudflare.com/t/urgent-malicious-requests-coming-from-cloudflare-workers-2a063600-103/809347>, section: Application Security [Cited on page 39.]
- [43] Cloudflare Workers (Jan 2026), <https://developers.cloudflare.com/workers/> [Cited on page 39.]
- [44] EFForg/cover-your-tracks (Feb 2026), <https://github.com/EFForg/cover-your-tracks>, original-date: 2015-11-02T20:23:32Z [Cited on page 23.]
- [45] Web crawler (Jan 2026), https://en.wikipedia.org/w/index.php?title=Web_crawler&oldid=1335488565, page Version ID: 1335488565 [Cited on page 5.]
- [46] Aghamohammadi, A., Eydgahi, A.: A novel defense mechanism against web crawlers intrusion. In: 2013 International Conference on Electronics, Computer and Computation (ICECCO). pp. 269–272 (Nov 2013) [Cited on page 20.]

- [47] Almahmoud, S., Hammo, B., Al-Shboul, B., Obeid, N.: A hybrid approach for identifying non-human traffic in online digital advertising. *Multimedia Tools and Applications* 81(2), 1685–1718 (Jan 2022) [Cited on pages 23, 25, and 44.]
- [48] Amin Azad, B., Starov, O., Laperdrix, P., Nikiforakis, N.: Web Runner 2049: Evaluating Third-Party Anti-bot Services. In: Maurice, C., Bilge, L., Stringhini, G., Neves, N. (eds.) *Detection of Intrusions and Malware, and Vulnerability Assessment*. pp. 135–159. Springer International Publishing, Cham (2020) [Cited on page 23.]
- [49] blocklist: www.blocklist.de – Fail2Ban-Reporting Service (we sent Reports from Attacks on Postfix, SSH, Apache-Attacks, Spambots, irc-Bots, Reg-Bots, DDos and more) from Fail2Ban via X-ARF., <https://www.blocklist.de/en/index.html> [Cited on page 15.]
- [50] Bolbol, N., Barhoom, T.: Mitigating Web Scrapers using Markup Randomization. In: *2021 Palestinian International Conference on Information and Communication Technology (PICICT)*. pp. 157–162 (Sep 2021) [Cited on page 21.]
- [51] Brach, W., Petrik, M., Košťál, K., Ries, M.: Ghosts in the Markup: Techniques to Fight Large Language Model-Powered Web Scrapers. In: *2025 37th Conference of Open Innovations Association (FRUCT)*. pp. 37–46 (May 2025) [Cited on page 21.]
- [52] Brewer, D., Li, K., Ramaswamy, L., Pu, C.: A Link Obfuscation Service to Detect Webbots. In: *2010 IEEE International Conference on Services Computing*. pp. 433–440 (Jul 2010) [Cited on pages 16 and 17.]
- [53] Davidson, A., Goldberg, I., Sullivan, N., Tankersley, G., Valsorda, F.: Privacy Pass: Bypassing Internet Challenges Anonymously. *Proceedings on Privacy Enhancing Technologies* 2018, 164–180 (Jun 2018) [Cited on page 18.]
- [54] Diab, A., Barhoum, T.: Prevent XPath and CSS Based Scrapers by Using Markup Randomizer [Cited on pages 7 and 20.]
- [55] Dikaiakos, M., Stassopoulou, A., Papageorgiou, L.: Characterizing Crawler Behavior from Web Server Access Logs. In: Bauknecht, K., Tjoa, A.M., Quirchmayr, G. (eds.) *E-Commerce and Web Technologies*. pp. 369–378. Springer, Berlin, Heidelberg (2003) [Cited on pages 11 and 38.]
- [56] Doran, D., Morillo, K., Gokhale, S.: A Comparison of Web Robot and Human Requests. pp. 1374–1380 (Aug 2013) [Cited on page 11.]
- [57] Haque, A., Singh, S.: Anti-scraping application development. In: *2015 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. pp. 869–874 (Aug 2015) [Cited on page 12.]
- [58] Howard, J.: The /llms.txt file (Sep 2024), <https://llmstxt.org/> [Cited on page 14.]

- [59] Istrate, R., Tulus, V., Grass, R.N., Vanbever, L., Stark, W.J., Guillén-Gosálbez, G.: The environmental sustainability of digital content consumption. *Nature Communications* 15(1), 3724 (May 2024) [Cited on page 1.]
- [60] Jiang, J., Song, X., Yu, N., Lin, C.Y.: FoCUS: Learning to Crawl Web Forums. *IEEE Transactions on Knowledge and Data Engineering* 25(6), 1293–1306 (Jun 2013) [Cited on page 5.]
- [61] Koster, M., Illyes, G., Zeller, H., Sassman, L.: Robots Exclusion Protocol. Request for Comments RFC 9309, Internet Engineering Task Force (Sep 2022), num Pages: 12 [Cited on page 11.]
- [62] Lee, Y.L., Hsu, C.H.: Usability study of text-based CAPTCHAs. *Displays* 32(2), 81–86 (Apr 2011) [Cited on page 18.]
- [63] Liu, E., Luo, E., Shan, S., Voelker, G.M., Zhao, B.Y., Savage, S.: Somesite I Used To Crawl: Awareness, Agency and Efficacy in Protecting Content Creators From AI Crawlers (May 2025), <http://arxiv.org/abs/2411.15091>, arXiv:2411.15091 [cs] [Cited on pages 9, 12, 13, 15, and 36.]
- [64] Longato, F.: LLMs.txt - Why Almost Every AI Crawler Ignores it as of August 2025 (Aug 2025), <https://www.longato.ch/llms-recommendation-2025-august/> [Cited on page 14.]
- [65] Malashin, I.P., Tynchenko, V.S., Gantimurov, A.P., Neluyb, V.A., Borodulin, A.S.: Detecting of Robotic Imitation of Human on-the-Website Activity With Advanced Vector Analysis and Fractional Derivatives. *IEEE Access* 12, 56707–56718 (2024) [Cited on page 26.]
- [66] Mckenna, S.F.: Detection And Classification Of Web Robots With Honeypots (Mar 2016) [Cited on pages 7, 15, 16, and 17.]
- [67] Miller, C.: ai.txt: A new way for websites to set permissions for AI (May 2023), <https://spawning.substack.com/p/aitxt-a-new-way-for-websites-to-set> [Cited on page 13.]
- [68] Rahman, R.U., Tomar, D.S., Kumar, P.: A polymorphic code generation engine with obfuscation techniques for protecting web bot attacks. *Iran Journal of Computer Science* (Jul 2025) [Cited on pages 18 and 21.]
- [69] Schwartz, B.: Google-Extended does not stop Google Search Generative Experience from using your site’s content (Oct 2023), <https://searchengineland.com/google-extended-does-not-stop-google-search-generative-experience-from-using-y> [Cited on page 9.]
- [70] See, R.A.: Emerging Threats to Online Security: Securing Systems Against Unauthorized Automation and Web Bots. doctoralThesis, Staats- und Universitätsbibliothek Hamburg Carl von Ossietzky (Mar 2025), accepted: 2025-07-24T12:31:46Z [Cited on pages 18, 23, 25, and 48.]

- [71] Server, C.W.: Caddy - The Ultimate Server with Automatic HTTPS, <https://caddyserver.com/> [Cited on page 36.]
- [72] Suchacka, G., Cabri, A., Rovetta, S., Masulli, F.: Efficient on-the-fly Web bot detection. *Knowledge-Based Systems* 223, 107074 (Jul 2021) [Cited on pages 24, 31, 48, 72, and 73.]
- [73] Tan, P.N., Kumar, V.: Discovery of Web Robot Sessions Based on their Navigational Patterns. *Data Mining and Knowledge Discovery* 6(1), 9–35 (Jan 2002) [Cited on pages 25, 31, 72, and 73.]
- [74] Tseriotis, A.M.: Advanced Web Scraping in the Modern Web [Cited on pages 11, 15, 17, 18, 19, and 27.]
- [75] Turk, K., Pastrana, S., Collier, B.: A tight scrape: methodological approaches to cybercrime research data collection in adversarial environments. In: 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). pp. 428–437 (Sep 2020) [Cited on pages 5, 6, 11, 12, 18, 19, and 27.]
- [76] Vastel, A., Rudametkin, W., Rouvoy, R., Blanc, X.: FP-Crawlers: Studying the Resilience of Browser Fingerprinting to Block Crawlers (Feb 2020) [Cited on pages 23 and 40.]
- [77] Ward, J.: How an Obscure CSS Feature Can Protect Your Art From AI (Feb 2026), https://medium.com/@joshward_accounts/how-an-obscure-css-feature-can-protect-your-art-from-ai-f5169ea5ff9a [Cited on page 22.]
- [78] WETTERSTRÖM, R., Andersson, S.: Web information scraping protection (Dec 2009), <https://patents.google.com/patent/W02009154564A1/en> [Cited on page 20.]
- [79] Yasuhara, K., Kodama, N., Saito, T.: Challenges in Web Bot Detection and Detection Evasion Technologies. In: Barolli, L. (ed.) *Advances in Network-Based Information Systems*. pp. 162–173. Springer Nature Switzerland, Cham (2024) [Cited on pages 15, 18, and 23.]
- [80] Zhao, J., Chen, R., Fan, P.: TS-Finder: privacy enhanced web crawler detection model using temporal-spatial access behaviors. *The Journal of Supercomputing* 80(12), 17400–17422 (Aug 2024) [Cited on page 25.]

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl