Ministry of Education and Science of the Republic of Kazakhstan
Suleyman Demirel University

Temirlan Kudabayev

Randomized and approximized algorithms

Presented in Partial Fulfillment for the
Degree of Master of Science in Computer Science
(degree code: 7M06102)
Department of Computer Sciences
Faculty of Engineering and Natural Sciences

Supervisor: **Meiram Murzabulatov**

Kaskelen, 2022

Suleyman Demirel University

Faculty of Engineering and Natural Sciences

Department of Computer Science

Dean of Faculty

Associate Professor, PhD Zhamanov A.

«20» 06 2022

Topic of the thesis:

Randomized and Approximized algorithms

Thesis submitted as part of the requirements for the award of the MSc in "7M06102 - Computer Science", SDU, 2020-2022

Head of Department Assos. prof. Cemil Turan

Academic Supervisor Meiram Murzabulatov

Master's student Temirlan Kudaibergen

Kaskelen, 2022

Abstract

In modern times, as the number of websites expands, so does the volume of data. Automated apps that can scan and extract the essential information, process it, and save it in a user-friendly format are in high demand among users. These programs are called web scrapers. The immense popularity of these applications compels online service owners to take additional steps to decrease the number of bots in their Internet service traffic. If Web Services Protection identifies the user as a bot, the server will block the user entirely. The primary objective of this master's thesis is to create a Node.js parser using the Selenium tool. In addition, in order to replicate human activities, our parser will utilize randomized algorithms. In this thesis, we will examine the work of server protection in greater detail; we will examine four common indicators by which the server identifies that the user is a parser, as well as the server's methods for preventing bots. We will analyze how using the Selenium tool and the introduction of randomized algorithms will help us bypass the blocking from the server. To obtain the results, we will parse five subcategories of the chosen website and evaluate the stability of our software based on four parameters: the number of pages parsed, the amount of data processed, the total number of errors and blocks, and the rate of data parsing per unit of time. In order to do a qualitative analysis, we will compare these indicators using the same parser but without the application of randomized algorithms.

Аңдатпа

Қазіргі уақытта веб-сайттар санының өсуімен деректер көлемі де артып келеді. Пайдаланушылар қажетті ақпаратты сканерлеп, шығарып алатын, оны өңдейтін және пайдалануға ыңғайлы форматта сақтай алатын автоматтандырылған қосымшаларға үлкен қажеттілікке ие. Бұл қосымшалар талдаушы болып табылады. Бұл қосымшалардың үлкен танымалдылығы веб-сервис иелерін өздерінің Интернет қызметтерінің трафигіндегі боттардың санын азайту үшін қосымша шаралар қабылдауға мәжбүр етеді. Егер веб-сервис қорғау қызметі пайдаланушының бот екенін анықтаса, сервер пайдаланушыны толығымен блоктайды. Бұл магистрлік жұмыстың негізгі мақсаты - Selenium құралын пайдаланып Node.js тілінде талдаушы жазу. Сондай-ақ, адам әрекеттерін имитациялау үшін біздің талдауышымызда рандомизацияланған алгоритмдер болады. Бұл мақалада біз серверді қорғау жұмысын егжей-тегжейлі қарастырамыз: біз сервер пайдаланушының талдаушы екенін анықтайтын төрт танымал белгілерді қарастырамыз, сонымен қатар сервердің боттарды блоктау әдістерін қарастырамыз. Selenium құралын пайдалану және рандомизацияланған алгоритмдерді енгізу сервердің қорғанысын айналып өтуге қалай көмектесетінін қарастырамыз. Нәтижелерді алу үшін таңдалған сайттың бес ішкі санатын талдаймыз және бағдарламамыздың тұрақтылығын төрт параметр бойынша тексереміз: талданған беттердің саны, талданған деректердің көлемі, қателер мен блоктардың жалпы саны және талдау жылдамдығы. Сапалы талдау үшін біз бұл көрсеткіштерді, рандомизацияланған алгоритмдерді жүзеге асырмай-ақ бір талдаушымен салыстырамыз.ады.

Аннотация

В настоящее время с увеличением количества веб-сайтов увеличивается объем данных. У пользователей существует огромная потребность в автоматизированных приложениях, которые могли бы сканировать и извлекать необходимую информацию, обрабатывать ее и сохранять в удобном для пользователя формате. Данные программы являются парсерами. Огромная популярность данных программ вынуждают владельцев веб-сервисов принимать дополнительные меры для уменьшения количества ботов в трафике своих интернет сервисов. Если защита веб-сервисов обнаруживает, что пользователь является ботом, то сервер полностью блокирует пользователя. Основная цель данной магистерской диссертации является написание парсера на языке Node.js, с использованием инструмента Selenium. Также для симуляции человеческих действии наш парсер будет иметь рандомизированные алгоритмы. В этой статье мы подробно рассмотрим работу защиты сервера: рассмотрим четыре популярных признака по которому сервер определяет что пользователь является парсером, а также рассмотрим методы сервера по блокировки ботов. Рассмотрим как с помощью инструмента Selenium и введения рандомизированных алгоритмов поможет нам в обходе блокировки от сервера. Для получения результатов, мы спарсим пять подкатегорий выбранного сайта и проверим стабильность работы нашей программы относительно четырех параметров: количество спарсенных страниц, количество спарсенных данных, общее количество ошибок и блоков, а также скорость парсинга данных в единицу времени. Для качественного анализа мы сравним данные показатели с тем же парсером, но без имплементации рандомизированных алгоритмов.

Acknowledgements

Thanks to my thesis supervisor for constant support and useful discussion. Thanks to the external reviewer for very useful feedback that helped to significantly improve the current work. Thanks to my family and especially to my spouse for all support.

To my family

Contents

Nomenclature

$AJAX$ Asynchronous JavaScript and XML

API Application Programming Interface

CSS Cascading Style Sheets

DOS Denial of Service

$HTML$ Hyper Text Markup Language

$HTTP$ Hypertext Transfer Protocol

$HTTPS$ Hypertext Transfer Protocol Secure

JS JavaScript

$JSON$ JavaScript Object Notatio

SEO Search Engine Optimization

$SOCKS$ Socket Secure

SPA Single Page Application

SSR Server Side Rendering

URL Uniform Resource Locator

$V8$ JavaScript and WebAssembly engine

$XPATH$ XML Path Language

1. Introduction

1.1 Motivation

With the introduction of the Internet, mankind's life was divided into two periods: before and after. The first website was developed in August 1991 by Tim-Berners Lee, a researcher at the European Center for Nuclear Research. It is still operational today [4]. The number of sites had expanded to ten by the end of 1992, and after the WorldWideWeb technology was released in 1993, the Internet began to grow fast, resulting in global changes. When Yahoo! started in 1994, the world had nearly 3,000 websites.

From January 2008 to January 2022, Table A.1 gives data on the number of sites [21]. According to this information, the number of sites reached 250 million in 2011 and over 1 billion sites were operating in 2016. There are 1.88 billion web pages on the Internet as of August 2021, and the number is growing. Today, the internet contains approximately 1.17 billion webpages [21]. Only 17% of these websites are active, while the other 83% are dormant. There are 200 million active websites, with 252,000 new ones being established every day.

Along with the rise in the number of websites comes an increase in the amount of unstructured data. There was a significant need among users for automated applications that could scan and extract the relevant information, process it, and save it in a user-friendly format. Parsers are the name for such applications. This isn't a brand-new technology. According to Gupta's research, parser robots are roughly the same age as the Internet [16].

Search engine robots are one of the most common scrapers. Yandex and Google are directly involved in parsing: they visit the site and index it, collecting data. This algorithm will aid in indexing the site correctly in relation to the search query. Scrapers are used by several companies, in addition to search engines, to

achieve a variety of goals:

- Market research data collection. Data mining web services can help you stay on top of where your company or industry is going, laying the groundwork for market research. Scraping software can get information from many different places and put it all in one place so it can be analyzed.

- For the purpose of price tracking. For specific sorts of items, parsing tools can be used to examine competitors' prices. Such analytics are required for enterprises to plan their own product development.

- To correct SEO mistakes. Technical site parsing is used to identify various site faults. They can use parsers to find pages that don't exist, pages that are duplicated, descriptions that aren't complete, products that don't have certain qualities, and stock balance data that doesn't match what's shown on the site.

All of the above factors are an incomplete list of how you can use parsing for your own purposes. That is why this tool is used by a huge number of businesses.

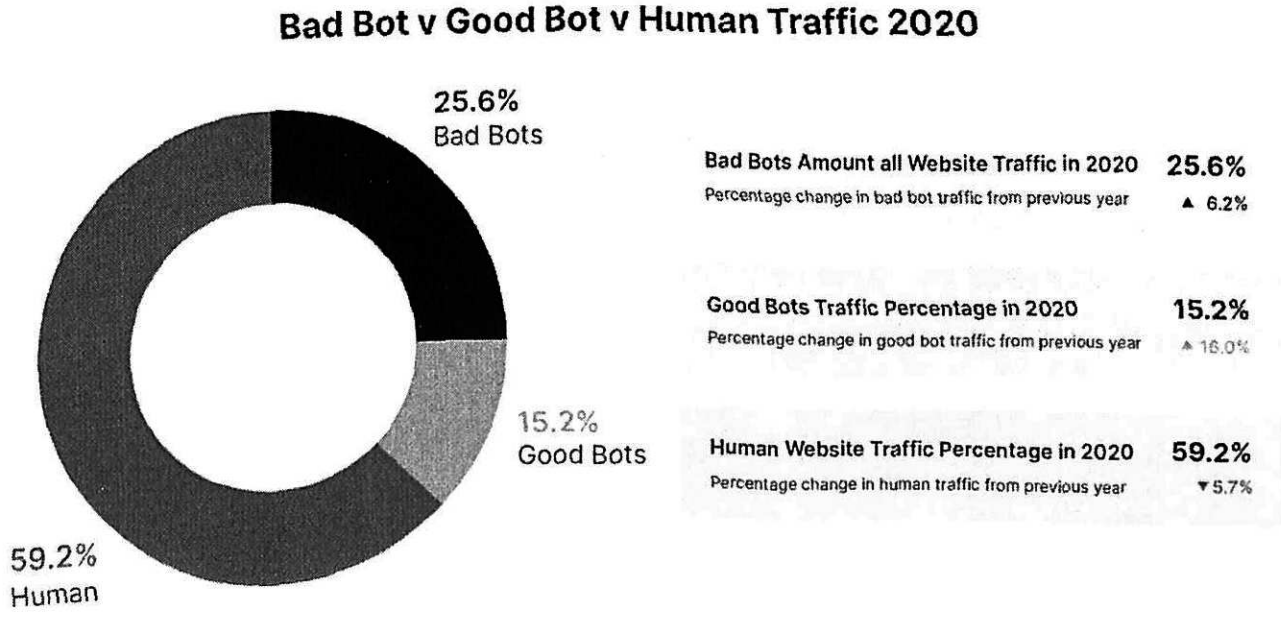

Figure 1.1: Traffic analyze in 2020 year

Figure 1.1 depicts an analysis of Internet traffic that revealed how much of it is occupied by bots and humans [18]. Humans create 59.2% of traffic, while bots

Good Bots Traffic Percentage in 2020	**15.2%**
Percentage change in good bot traffic from previous year	▲ 16.9%
Human Website Traffic Percentage in 2020	**59.2%**
Percentage change in human traffic from previous year	▼ 5.7%

generate the rest, according to the report. Bots are also classified as either bad or good. Bad bots, on the other hand, attempt to harm the web resource. Such bots can launch DoS assaults and hack a variety of data sources, violating the stability of the web resource. Web scrapers and other bots are examples of good bots.

Web service owners set up extra measures to reduce the number of bots in the traffic of their Internet services. If it detects that the user is a bot, it evaluates their behaviors and prevents access to the website regardless of whether they are good or negative.

1.2 Aims and Objectives

In this article, we will use the server-side Node.js language to build our own parser system. For the parsing library, the Selenium Webdriver tool will be used. This software is often used by data parsers and test developers. The design of a parser that will not interfere with the website's stability is a vital feature.We will use random algorithms to act like people and keep the web service from getting too busy. It's also important to note that this parser won't break any copyright laws and won't get any personal or confidential information from users.

1.3 Thesis Outline

This dissertation will be separated into logical sections for the best structure. The first chapter is Introduction chapter. It is this one that you are currently reading. The rest of the paper is laid out as follows. The literature review is introduced in Section 2. Our methods and materials are presented in Section 3. Section 4 contains the data and results. Finally, in Conclusion chapter we conclude our conclusion.

2. Literature Review

Similar articles or publications related to parsing or constructing a parser from scratch will be described in this area. A large number of minor articles, manuals, or training resources were discovered when searching for information. Due to their modest size, many types of materials were overlooked. Four good pieces were discovered throughout the search:

- Xurxo Legaspi, "Scraping Dynamic Websites for Economical Data" [26].
- Huy Phan, "Building Application Powered by Web Scraping" [31].
- Neal R Haddaway, "The Use of Web-scraping Software in Searching for Grey Literature" [17].
- Javier Ochoa Serna, "Design and Implementation of a scraping system for sport news" [33].

These projec ts demonstrate a variety of approaches to utilize the parser to achieve their objectives. Each of these articles will be placed in its own subcategory, with links to the full versions of these articles in the references section. This section's final subcategory will have output.

2.1 The importance of parsing in the processing of various types of open data

Scraping websites is the most effective approach to automate the process of gathering and saving data. You can use the parser to make and maintain webpages with identical design, content, and structure. Xurxo Legaspi (2016) wrote a paper about parsing sites with dynamically changing content that employ AJAX

technology to update dynamic data on a regular basis. Sites having a slant in economic data, for example, where a large number of charts are used. The author suggests parsing with the LnuDSC (Lnu Data Stream Center) auxiliary tool and the Scrapy library [26]. Neal R Haddaway (2015), on the other hand, emphasizes the challenge of locating gray literature for research activities like systematic reviews. The author concentrates on the parser as a tool for solving the search for gray literature, which is a unique aspect of this essay [17]. Javier Ochoa Serna (2017) developed a program to extract data from a news page about popular Spanish football teams [33]. The author of the article evaluated the available data and created a web dashboard for easy examination of the study results thanks to the development of an unique parsing tool. Javier Ochoa Serna (2017) visualized the data by separating the analytical data into different columns for each of the news stories. In terms of concept, this piece is really intriguing. This article's author was able to condense a lengthy development process into a single structure. It is important to mention that the building of a parser is simply one component of his endeavor in his article, not the full dissertation's purpose.

2.2 Security Bypass Methods on the Server

In the techniques section, I discussed common indicators that the server recognizes a site visitor as a parser, as well as several types of blocking and how to get around them using the Selenium tool and randomized algorithms. Huy Phan (2019) also provides a technique from the ground up that allows you to parse data from web pages of websites in his work. Section 2.6.4 Detect anti-bot mechanism and halt, which is dedicated to the introduction to the server protection problem, explains how one of the server protection methods works by analyzing human activities [31]. When examining a web site, the server protection puts it into one of three categories: white list, gray list, or black list, according to the edition of this article. The number of requests sent by the client, the load, and digital fingerprints all influence this process. The article's author also highlights that the parser need continual developer assistance owing to constantly changing data and the constant update of old sites. This article is the most similar to my subject. Huy Phan's (2019) essay takes a unique approach to developing the parser and writing instructions on how he constructed it from the ground up [31]. My post,

on the other hand, goes into great depth about how to protect the server from parsing.

2.3 Conclusion

After reviewing the aforementioned articles and dissertation, it is important to note that all of the authors underlined the relevance of web scrapers as a technology. Web scrapers, according to most, assist in the organizing and cleaning up of data in loose, unstructured collections of information. According to an article by Huy Phan (2019), as the economy shifts toward a high-tech business, getting the data you need in a reliable and timely manner becomes increasingly vital. Parsers have a lot of room for improvement as a tool for navigating large, unstructured data sets [31]. In addition, all of the authors underlined the relevance of data that has been copyrighted. This means that before analyzing the data, you should familiarize yourself with the site's copyright policy. Furthermore, each of the aforementioned authors used parsing for completely different reasons, ranging from designing the parser to analyzing the data collected by the parser. Unlike the other papers, the purpose of my research is to create a parser that, in addition to collecting data, simulates human activities using randomized algorithms, overcoming the parser's protection and causing no disruption to the web resource.

3. Methods and Materials

Developing any algorithm from scratch is a fairly voluminous task. In order to cope with this task, it is necessary to divide it into several parts. All these parts will be logically separated with respect to their task.

1. Learn about website security systems.
2. Learn about the Selenium parsing library.
3. Learn about randomized algorithms.
4. There are ways to bypass website security systems using Selenium and randomized algorithms.
5. Choose a website with strong protection against parsing.
6. Writing a program in Node.js using the Selenium library
7. Implement randomized algorithms in the program base.
8. Test the work of the parser with randomized algorithms and without randomized algorithms.

The first four parts of this plan are the theoretical part of this study. This was necessary to create a stable knowledge base. The rest of parts five through eight inclusive are practical. All parts of this plan will be described in more detail below.

3.1 Learn theory about websites security system

A large number of such programs, written individually for different customers, daily visited the sites of online stores and opened many pages with goods. Since

many of the programs were not written by professionals, they sent a large number of requests to the servers and created a significant load on the sites, slowing down their work or even potentially leading to the unavailability of the resource. This state of affairs did not suit the owners of large online stores, and special tools began to appear on the market to protect against parsing-systems for protecting against bots.

Parsing protection means that it will be as difficult as possible for scripts and bots to get data from your site (online store), while access to the site for real users and search engines will not be compromised. Unfortunately, this is a rather difficult task because it is necessary to find a compromise between protection from parsers and the complexity of access for real users and search engines.

In order to fully cover the topic of protection against parsing, it is necessary to understand how the server detects the parser and how the server blocks the parser. At the moment, there are a large number of ways to determine the parsing server: The pattern in sending requests, browser fingerprint reader, lack of JavaScript/CSS rendering and trap link for bots [24].

3.1.1 The pattern in sending requests

People in general don't stick to repetitive activities because they visit sites in different ways. If they don't have any features, data collection bots usually have a place in crawling and collecting data because they are coded that way. Sites with smart data collection protection can easily find spiders because they can see their activity and stop them from cleaning up data.

Data cleansing bots collect data very quickly. People can't surf the web at that speed, so the site could easily be owned by your scraper. The protection recognizes it as a server faster the faster the parser searches for and collects data.

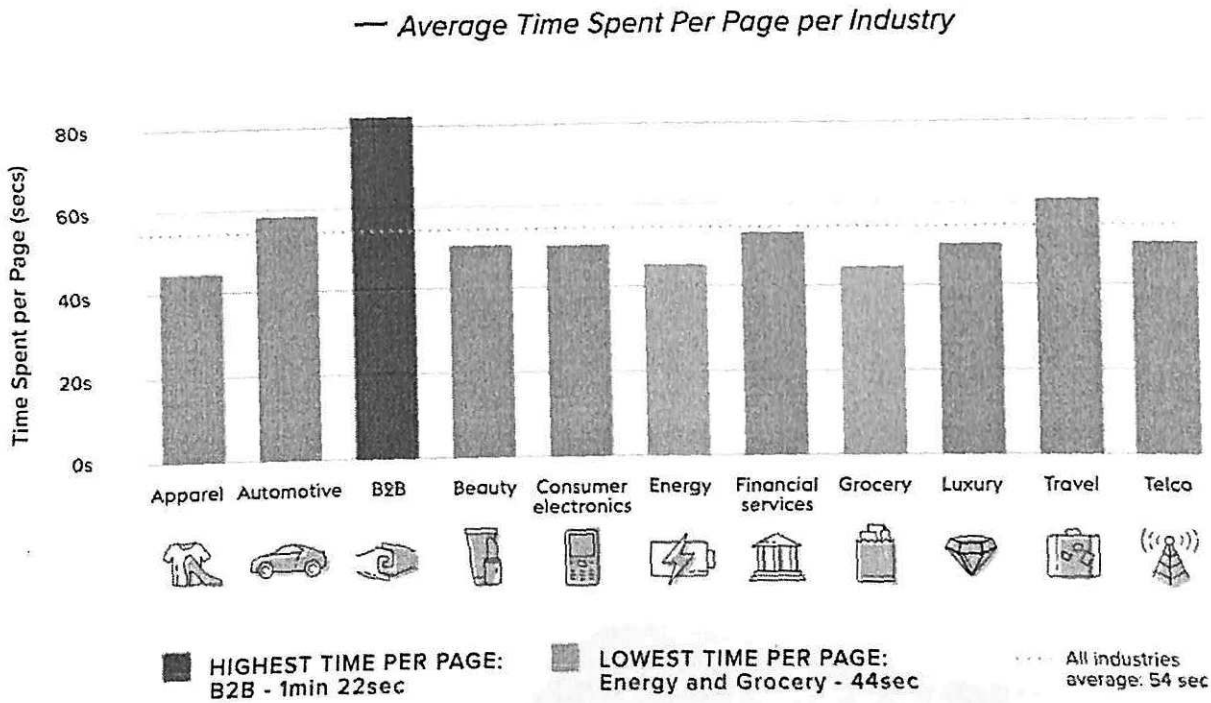

Figure 3.1: Average time spent per page per industry.

Figure 3.1 shows the analytics of the average time spent per page per industry [14]. The data shows that the average person spends 54 seconds on a page.

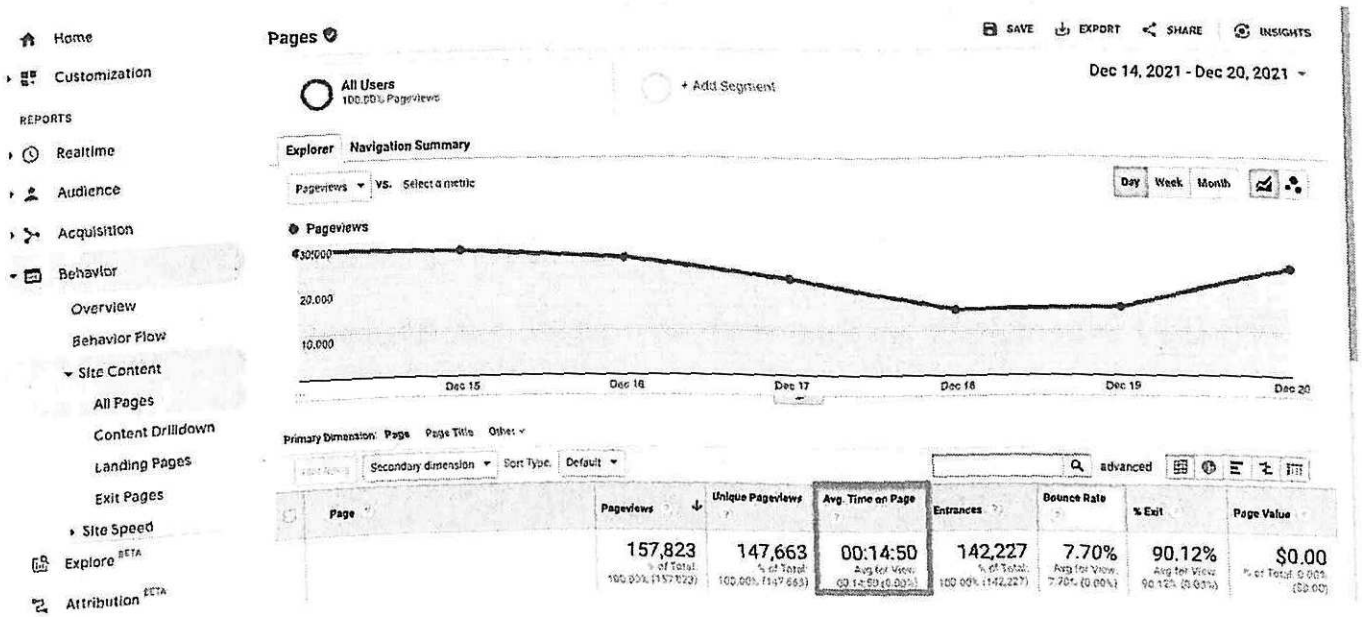

Figure 3.2: Google Analytics.

Figure 3.2 shows the analytics from Google Analytics regarding the average time spent on the page per user. Figure 3.1 and Figure 3.2 were taken from the same resource [14]. This data shows that, on average, a person lingers on a page for a certain amount of time.

3.1.2 Browser fingerprint reader

Server protection can take various data from the client, including its digital fingerprints. One of these fingerprints is the User-Agent parameter, which the browser automatically generates when working with websites. The User-Agent request header is a string that tells servers and other network peers about the application, operating system, vendor, and/or version of the User-Agent making the request [11].

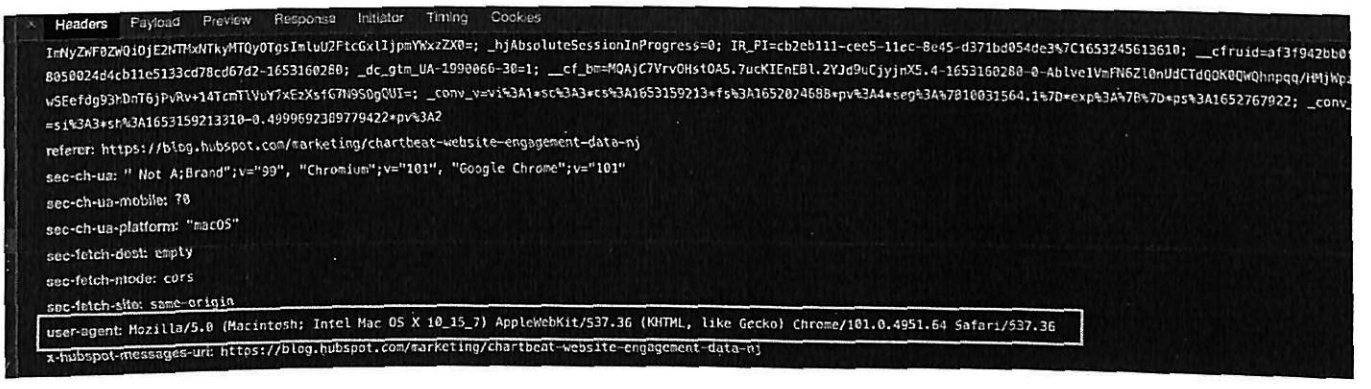

Figure 3.3: User-agent parameter.

Figure3.3 shows the automatic generation of the User-Agent parameter. This information helps the server collect its own analytics or keep track of events regarding a single user of the website. The absence of this parameter in the protection of the website causes great suspicion.

3.1.3 Lack of JavaScript / CSS rendering

The real browser requests and loads resources such as images and CSS. HTML parsers and crawlers will not do this, as they are only interested in the actual pages and their content.

Figure 3.4: The loading process of JavaScript files.

The loading of JavaScript files for the Google search engine can be seen in Figure 3.4. If the site client does not load JavaScript, then the site actually marks such a visitor as a bot. While it is possible to prevent JavaScript from running in a browser, almost all sites on the Internet will be unusable, which means that this feature will be enabled in most browsers.

The React.js library is used by 40.14 percent of 67.593 respondents [36] in the "Stack Overflow Developer Survey 2022." This statistics indicates that the use of front-end frameworks and libraries is becoming more popular. The use of the React library results in the building of a fully functional Single Page Application. A single-page application (SPA) is a single page that interacts with the user on a continuous basis by dynamically rewriting the current page rather than downloading complete new pages from the server. A single block tag is supplied on the main HTML page, to which JavaScript programs are attached. Because of this, you need to be able to load Javascript to get the full content of an SPA application.

3.1.4 Bot Trap Link

The work of many parsers is to collect information from the desired page and go to the next one. Usually, the parser algorithm collects all the links on the pages and then follows them to further collect information. In order to counteract this method of parsing, decoy links were invented. This method is one of the easiest to implement.

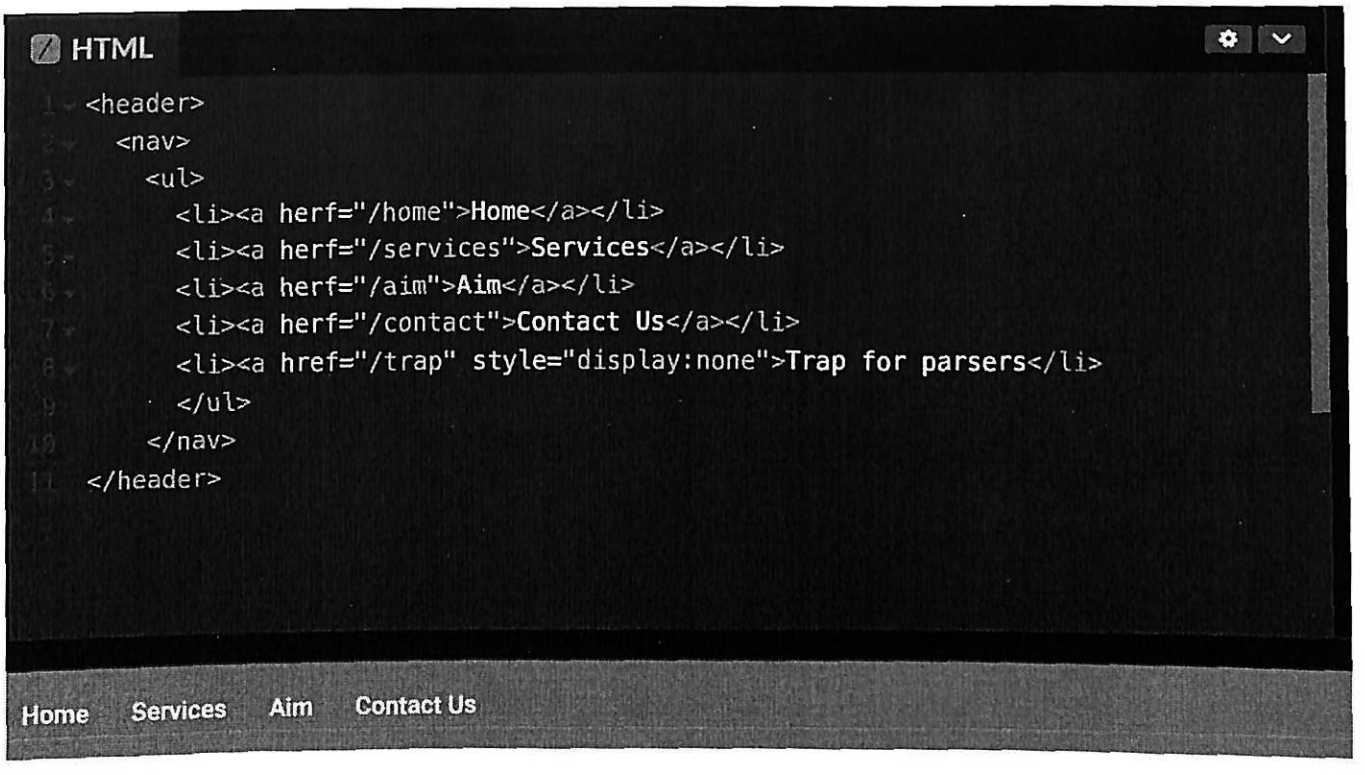

Figure 3.5: A simple trap for scraper bots.

Figure 3.5 shows the simplest implementation of a hook for parsers. Usually, they are made invisible to the user or given to a user who shows suspicious activity.

When a decoy link is clicked, the website blocks the user or slows down the user's response time. The main disadvantage of traps is that those who parse know about them and find them after some time. And when the developers know where the traps are, they set up the parser so that it avoids them. For the traps to work effectively, their url and location on the site need to be changed periodically.

After considering the signs by which the server protection can detect the parser, it is necessary to understand what measures the protection will take to neutralize or destabilize the parser. In this step, we will consider the four most popular solutions: Complete blocking by IP address, text-to-image conversion, data spoofing, changing the server data structure and Captcha [29].

3.1.5 Complete blocking by IP address

An IP address is a string of digits separated by periods. An IP address is made up of four integers, for example, 192.168.1.31. The set's numbers can vary from 0 to 255. As a result, the entire IP addressing range is 0.0.0.0 to 255.255.255 [25]. An IP address is a unique identifier that allows data to be exchanged between devices on a network; it contains information about the device's location and makes it

available for communication. IP addresses are used to identify computers, routers, and websites on the Internet and are an important part of how they operate.

IP blocking is one way to block access to a network resource by a client that has a specific IP address (or an IP address from a given set). This method is ineffective when denying access from one specific IP address. Because, often, IP-addresses are issued automatically when the client connects to the network. And its effectiveness increases when using a subnet mask. because getting online from another network is much more difficult. However, blocking a single user based on their subnet mask is unacceptable as this will affect other clients on the network.

```php
class RestrictIpAddressMiddleware
{
    // Blocked IP addresses
    public $restrictedIp = ['192.168.0.1', '202.173.125.72', '192.168.0.3', '202.173.125.71'];

    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        if (in_array($request->ip(), $this->restrictedIp)) {
            return response()->json(['message' => "You are not allowed to access this site."]);
        }
        return $next($request);
    }
}
```

Figure 3.6: An example of code that blocks by IP address

The algorithm for blocking an IP address is quite simple. After detecting a program or a bot and a server, the program reads the IP address and adds it to the blacklist. Figure 3.6 shows an example of blocking by IP address in PHP using the Laravel framework [28]. Despite the blocking, after connecting to another network or rebooting the modem, the user's IP address will change. After this operation, the client can also re-enter the blocked site.

3.1.6 Convert text to image

The server can convert the text to an image on the server and use it for display, which will make it difficult for simple parsers to extract the text. This method is

mediocre for a number of reasons:

- First, this method does not work well with screen readers. Due to the lack of content availability, this method will worsen the perception of content by a certain number of users [12].

- Secondly, due to the way search parsers work, this solution will not work well with search algorithms.

- Thirdly, this protection method can be bypassed using Optical Character Recognition programs [5]. These programs allow you to read text from an image. They take in an input image, the text inside of which must be read, and give the output text that was inside the image.

Despite all the above factors that significantly impair the performance of the website, this protection is widely popular.

3.1.7 Data replacing

After detecting the parser, the server can replace the real content of the website with a fake one. Instead of a real description of the goods, lorem ipsum [30] may be used. Or, instead of the real price of the product of 50, $theparserreceivesapriceof5($ This substitution of data for fake significantly worsens the data that the parser receives. As a result, incorrect or inappropriate statistics may be generated. This countermeasure is a very effective solution to parsing. The key feature of this protection is the difficulty in detecting fake content.

There are various more methods for altering data. Number parsing protection is used by the popular Avito marketplace service [19]. This is a free service that guards you from unsolicited phone calls and messages. It keeps user contacts private; they don't fall into the hands of scammers, and they don't get spam via SMS or instant messenger. Suspicious phone numbers are automatically rejected. When a buyer hits the "Show phone" option on a seller's offer, the service is displayed instead of the genuine phone number. When someone calls it, the service transfers the call and informs them that they are calling from Avito.

3.1.8 Changing the site structure

Parsers adjust to a specific site structure, and if it changes, the program crashes. Parsers understand the structure of the site's content using CSS styles, HTML structure, or XPATH. Therefore, programmers, in order to hinder the work of competitor parsing programs, can change the locations of components on the website, the names of classes, styles, etc. In order for this method of protection against parsing to work, you need to change the structure of the site regularly. The more often, the better.

The site structure is sometimes changed dynamically, i.e., when you go from different places to the same link, the content of the page will be different. This is often used in advertising, but it can also be used for protection. However, this method must be used with caution. The fact is that for SEO promotion, the site structure is a key component [3].

3.1.9 Captcha

A captcha is a specific security code that appears as a pop-up window or an image on some websites. The user is asked to perform a simple activity, confirm the status, type in words or figures, or respond to a question. The user will be unable to continue using the site if he or she does not confirm or answer the job. The application identifies whether the website's client is a person or a server in this way.

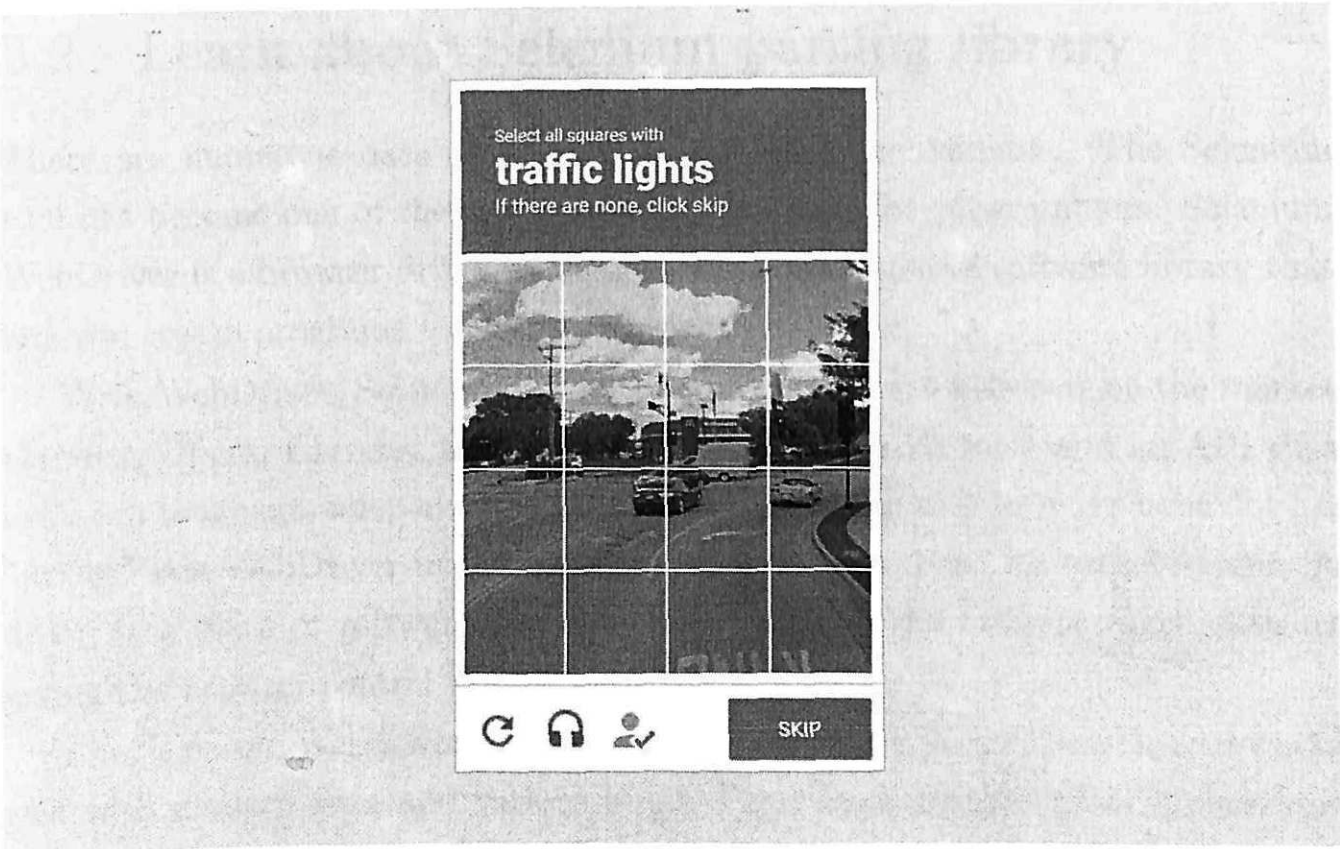

Figure 3.7: Captcha with the ability to select specific zones

Figure 3.7 shows a captcha with a selection of the desired images. In this figure, you need to select traffic lights. It should be noted that some parts of these pictures can be located on two blocks. This makes it very difficult for various programs to solve this captcha. In addition to the type of captcha shown in Figure 3.7, there are other types: Entering text, numbers, and symbols Solving a mathematical problem Confirming an action and others.

To work with Captcha, you need to connect a script from Google [1]. Once initialized with the website form, Captcha will successfully sync with the website. There are some services for bypassing Captcha-the most popular of them is the AntiCaptcha service [27]. However, it is worth noting that most of the stable-working services for solving Captcha are paid. For example, the AntiCaptcha service charges 0.5 USD for each 1000 captchas, depending on your volume [27].

We have considered five popular ways of server operation by blocking the stability of the parser. In practice, these methods can be much more. All of the above factors will only be obstructed if the server understands that it is the parser that is working. To improve our work, it is necessary to beware of the signs by which the parser can give itself away.

3.2 Learn about Selenium parsing library

There are numerous data parsing tools and libraries available. The Selenium tool has become one of the most popular such tools for programmers. Selenium WebDriver is a browser driver by design, which means it's a software library that lets you create programs that control browser behavior.

With WebDriver, Selenium can automate all the main browsers on the market (Firefox, Opera, Chrome, and Edge). WebDriver is a protocol and an API that defines a language-independent interface for managing web browser behavior. A "driver" is a WebDriver implementation that is customized for each browser. A driver is a piece of software that lets Selenium and the browser share data by giving the browser control [37].

The Selenium Webdriver tool belongs to the browser parsers, so this tool works well with modern sites and various Single Page Applications. SPA applications generate content using JavaScript and script parsers will not be able to work with them. It should be noted that now there is a trend toward using SSR applications or the introduction of special frameworks to speed up SPA applications. These techniques greatly influence the final result of the web application structure. SSR applications work well with SEO and are also good for speeding up website loading. Tools such as Next.js were created specifically to speed up the loading of a React.js application [23]. Even so, the Server Side Rendering feature lets you hide the execution of public API requests, which can cause problems when scripted parsers are being used.

In addition to all the above features of this tool, there are also the following advantages of using this tool:

- This tool supports different programming languages. This tool can work with languages such as Java, C, PHP, Ruby, Perl, Python, and Node.js, which means that a large number of developers can use it.
- It is open source and free for any developer.
- A large community of users - in case of difficulties in the work, there is someone to ask for help.

A large community of users - in case of difficulties in the work, there is someone to ask for help. Apart from data parsing, Selenium Webdriver is used in web site

testing. Testers use Selenium to automate the actions of browsers, check the functionality of the program and receive data from sites. To test how this tool works in practice, we will create a small function, presented below. This function will launch an automated Chrome browser, go to the Google page, write the word "Webdriver" in the text field, and go to the result page.

```
const webdriver = require('selenium-webdriver');
const chrome = require('selenium-webdriver/chrome');
const chromedriver = require('chromedriver');

chrome.setDefaultService(new chrome.ServiceBuilder(chromedriver.path).build());

const driver = new webdriver.Builder()
    .withCapabilities(webdriver.Capabilities.chrome()) !Builder
    .build();

(async function googleExample() {
    try {
        await driver.get('http://www.google.com/ncr');

        await driver
            .findElement(webdriver.By.name( name: 'q'))
            .sendKeys('webdriver', webdriver.Key.RETURN);
    } finally {
        await driver.quit();
    }
})();
```

Figure 3.8: Demonstration of the code of a standard Selenium application.

Figure 3.8 shows the code for the googleExample function that launches the automated Google Chrome browser. It's important to remember that the driver is initialized before the main function begins.

Figure 3.9: The HTML code for the Google homepage.

Figure 3.9 shows a small section of the Google page's HTML code. Among this code, a specific input attribute with the name=q parameter is highlighted. Figure 3.9 shows how WebDriver uses the findElement method to find the input element that is on the Google homepage. Next, using the sendKeys function, the browser writes the webdriver text and then calls the Enter button to go to the result page.

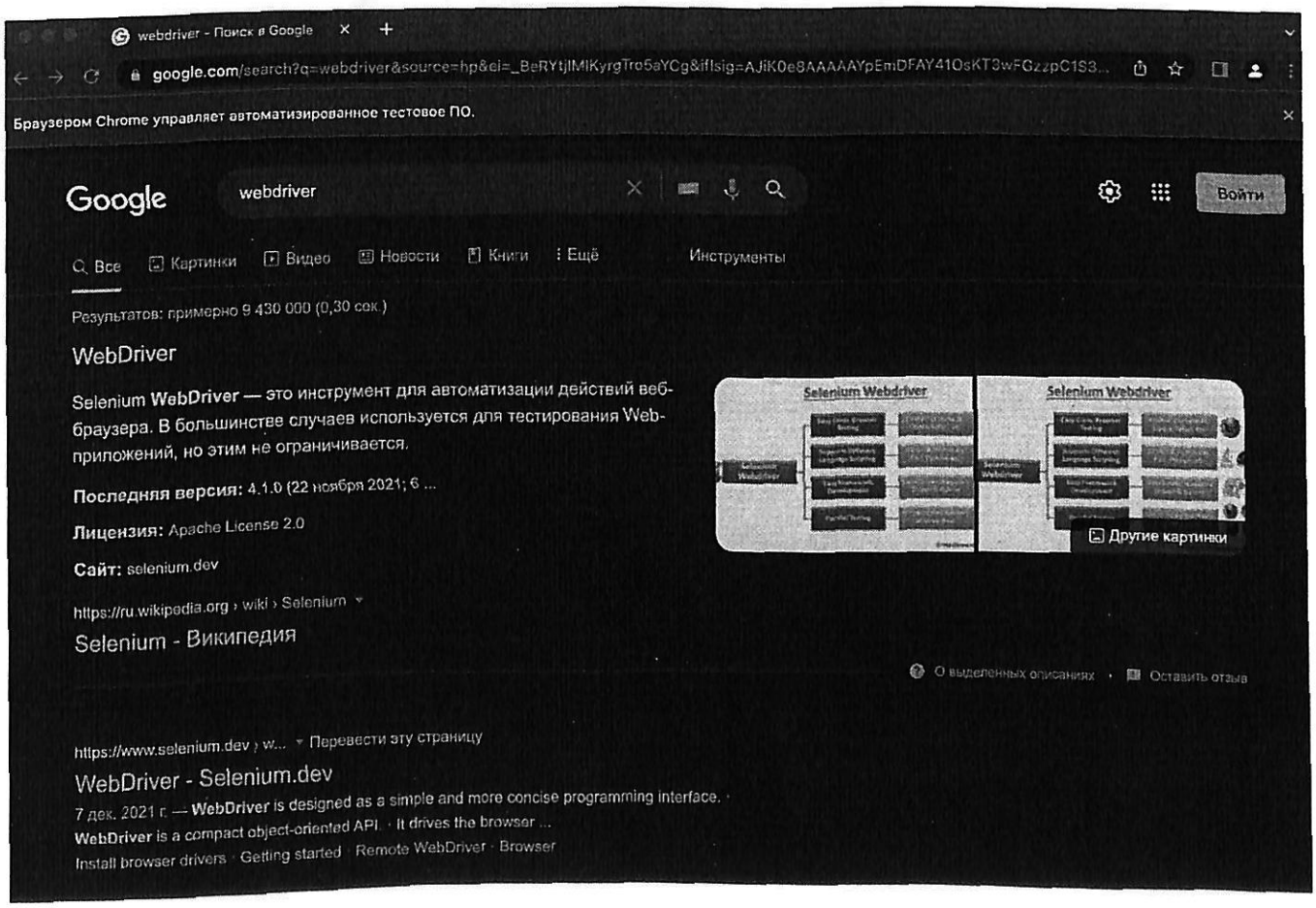

Figure 3.10: Selenium Webdriver in action.

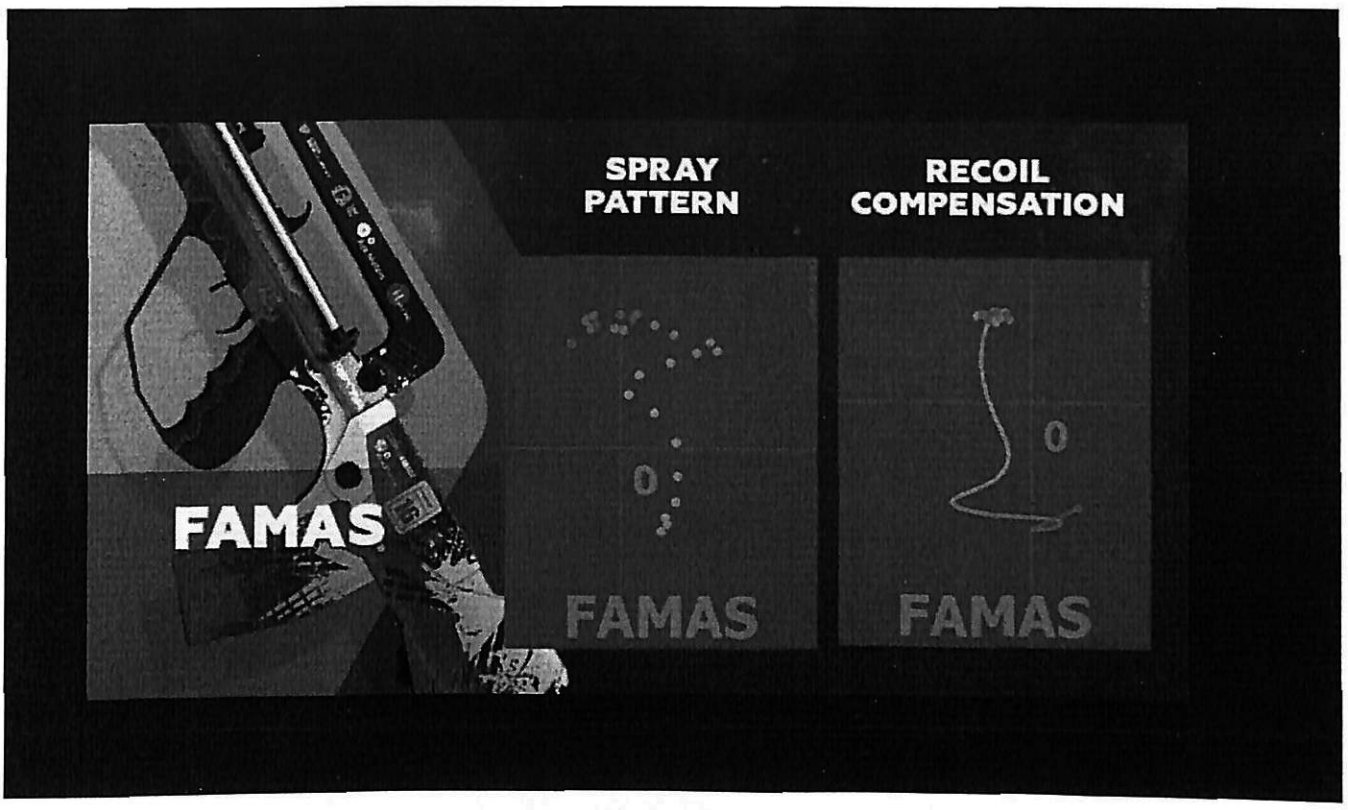

Figure 3.11: Famas weapon recoil simulation.

Figure 3.10 shows the end result of the googleExample function. It should be noted that the system warns that this browser works according to a certain algorithm. Figure 3.10 also shows an alert that sits between the input line and the Google page. This alert says that "Chrome Browser is managed by automated test software".

The Selenium webdriver allows programmers to control all browser features. The above example with the googleExample function shows that we can navigate to different pages, work with elements on the page, wait for the page to load, and much more. These above factors are a good reason to use Selenium Webdriver as a scraping tool.

3.3 Learn about randomized algorithms

Randomized algorithms are algorithms that use random as part of their program or base. I.e., when using randomized algorithms at some stage, instead of making a decision ourselves, we call on fate (chance) to choose for us. Randomness has many applications in science, art, statistics, cryptography, games, gambling, and other fields [23].

```
uint64_t state0 = 1;
uint64_t state1 = 2;
uint64_t xorshift128plus() {
  uint64_t s1 = state0;
  uint64_t s0 = state1;
  state0 = s0;
  s1 ^= s1 << 23;
  s1 ^= s1 >> 17;
  s1 ^= s0;
  s1 ^= s0 >> 26;
  state1 = s1;
  return state0 + state1;
}
```

Figure 3.12: xorshift128+ algorithm code.

The recoil of weapons in online shooters is an example of how randomized algorithms are used in games. The precise trajectory of a bullet after a shot is determined by a large number of variables. They use random numbers to avoid having to calculate each factor. Figure 3.11, for example, is a recoil simulation of a Famas weapon in the popular game Counter-Strike: Global Offensive. Bullets are fired into the center, rising slightly upwards, until the 15th. The shooting goes right and left at random after the 15th bullet [2]. This method makes it much easier for programmers to complete their tasks and calculate the bullet trajectory algorithm.

Despite the huge popularity of using randomness in various fields of science and technology, recreating absolute randomness is a very difficult task. In programming, pseudo-random number generators are used to recreate randomness. A pseudo-random number generator is an algorithm that generates a sequence of numbers whose elements are almost independent of each other and obey a given distribution [32]. At the moment, there are a huge number of such generators. For example, the people who make browser engines choose which algorithm for pseudo-random numbers will be used in their environment.

```
uint64_t state0 = 1;
uint64_t state1 = 2;
uint64_t xorshift128plus() {
  uint64_t s1 = state0;
  uint64_t s0 = state1;
  state0 = s0;
  s1 ^= s1 << 23;
  s1 ^= s1 >> 17;
  s1 ^= s0;
  s1 ^= s0 >> 26;
  state1 = s1;
  return state0 + state1;
}
```

Figure 3.12: xorshift128+ algorithm code.

The recoil of weapons in online shooters is an example of how randomized algorithms are used in games. The precise trajectory of a bullet after a shot is determined by a large number of variables. They use random numbers to avoid having to calculate each factor. Figure 3.11, for example, is a recoil simulation of a Famas weapon in the popular game Counter-Strike: Global Offensive. Bullets are fired into the center, rising slightly upwards, until the 15th. The shooting goes right and left at random after the 15th bullet [2]. This method makes it much easier for programmers to complete their tasks and calculate the bullet trajectory algorithm.

Despite the huge popularity of using randomness in various fields of science and technology, recreating absolute randomness is a very difficult task. In programming, pseudo-random number generators are used to recreate randomness. A pseudo-random number generator is an algorithm that generates a sequence of numbers whose elements are almost independent of each other and obey a given distribution [32]. At the moment, there are a huge number of such generators. For example, the people who make browser engines choose which algorithm for pseudo-random numbers will be used in their environment.

Figure 3.12 shows the xorshift128+ pseudo-random number generator code. It uses 128 bits of internal state, has a period length of 2128-1, and passes all the tests from the TestU01 suite. Since 2015, most browser engine developers have used this algorithm. This list includes Firefox, Safari, and browsers using the V8 engine (Chromium-like browsers) [15].

3.4 Ways to bypass website security systems using Selenium and randomized algorithms.

In order to understand how randomized algorithms can help bypass server protection, it is necessary to consider each of the parser protections separately. In this case, we will consider the signs from paragraph 3.1.1 to ?? inclusive, as well as the blocking itself from paragraph 3.1.5 to 3.1.9 inclusive.

3.4.1 Ways to Bypass Parsing Signs with Selenium and Randomized Algorithms

In total, from paragraph 3.1.1 to ?? inclusive, we considered four popular signs by which server protection can understand that a site user is a scraper. From Section 3.4.1 through Section 3.4.1, we will look at how Selenium tools and the use of randomized algorithms can help with feature traversal.

Bypassing the feature "The pattern in sending requests".

The pattern in sending requests says that the server, for certain reasons, can understand that this website client is a parser and not a person. The main indicator that a website client is a parser is the speed of sending data.

In order to recreate human activity, the program needs to set a speed limit. This operation is easily done using special wait functions. These asynchronous functions will temporarily stop the program for a certain amount of time to simulate human reading. It is worth noting that this timer should return a different wait value each time. In this case, it will help us to make randomized algorithms. Randomized algorithms will help us choose a time span from A to B, where A is the minimum wait time and B is the maximum wait time, thus choosing a unique

time span each time. Due to a random timer, web server protection will not be able to see a pattern in the way requests are sent.

In addition to setting randomized timers, using the Selenium tool, we can set up a function that will randomly click on some empty parts of the website. Since human actions are not systematic and not ideal, this function will help in recreating human actions.

Bypassing the "Browser fingerprint reader" feature.

Browser fingerprinting indicates that the server very often calculates browser fingerprints. Browser fingerprints such as the User-Agent parameter are used by websites for their own analytics. The User-Agent parameter, like many other browser fingerprints, is automatically recreated by browsers. The Selenium Webdriver tool launches a pre-programmed browser that is exactly the same as other browsers people use. The presence of this tool means that it is very easy to bypass this feature, since the programmed browser will automatically create all the necessary digital fingerprints.

Bypassing the "Lack of JavaScript / CSS rendering" feature

The lack of a JavaScript or CSS rendering flag means that the server can keep track of whether the client is loading Javascript or CSS files or not. Using the Selenium Webdriver tool, the programmed browser will automatically follow all the necessary links and download all the website's content.

Bypassing the "Link-trap for bots" feature

The link-trap feature for bots means that programmers create special links for bots, following which the server will understand that this client is a parser. In order not to fall into this trap, we must follow only visible links. The program needs to check styles and link classes for "display: none" styles.

Based on the above factors and how to bypass the four signs of parsing, we can say that using the Selenium tool and randomized algorithms is a good combination for solving this problem.

3.4.2 Ways to bypass locks with Selenium and randomized algorithms

In total, from paragraph 3.1.5 to 3.1.9 inclusive, we have considered four popular ways to block the parser. From Section 3.4.2 to Section 3.4.2, we will look at how Selenium tools and the use of randomized algorithms can help in bypassing locks.

Bypass blocking "Complete blocking by IP address".

When a user is completely blocked by an IP address, the user does not have client access to the website. In this case, in order to bypass this blocking, you should use a proxy server.

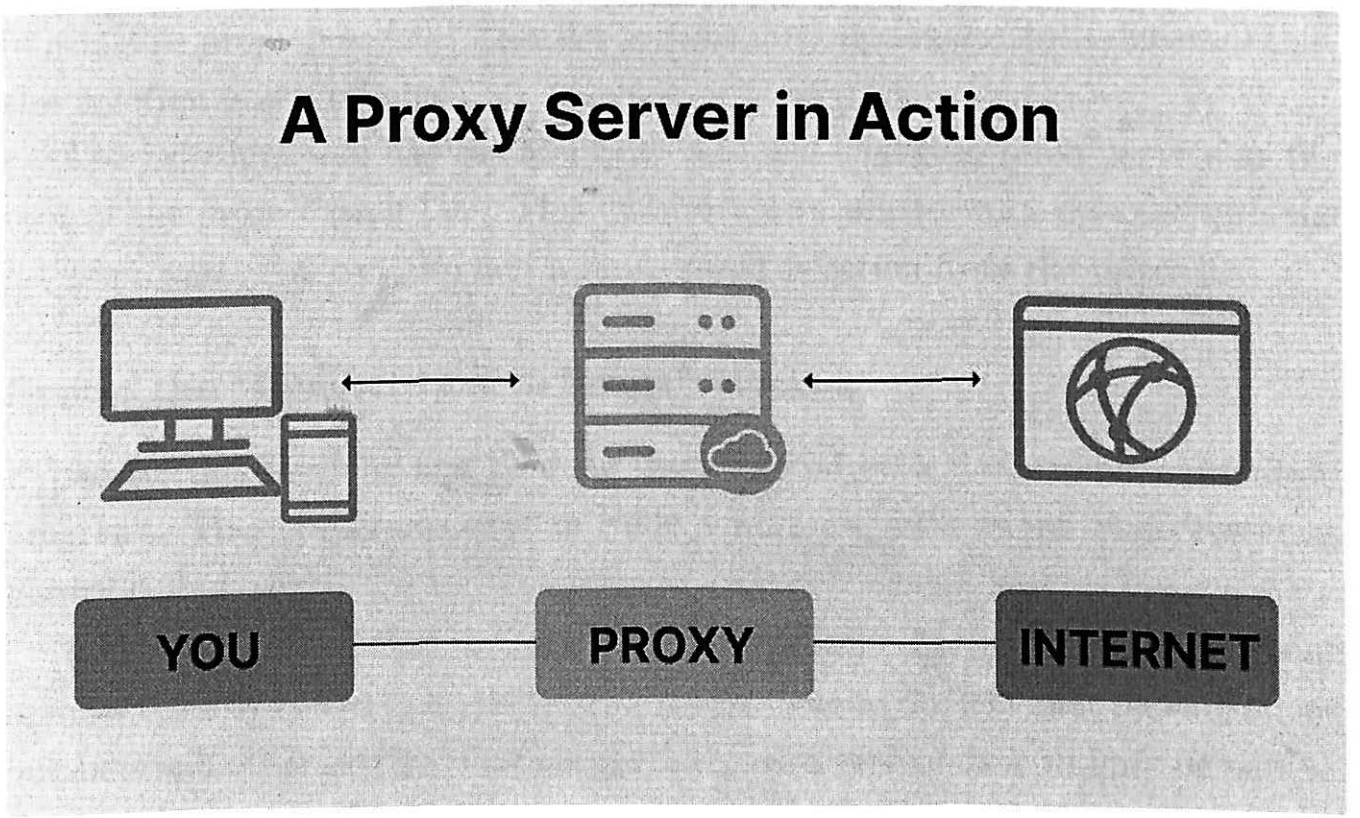

Figure 3.13: Proxy server operation in practice.

A proxy server is an intermediate server between the Internet user and the servers from which information is requested. Figure 3.13 shows an example of how a proxy server works in practice. The proxy server changes the user's IP address, thereby bypassing the blocking by IP address. The web server then sends a response containing the website data directly back to the user [6]. As a result, using a proxy server allows you to be on the Internet on behalf of another

server. Due to the fact that servers cannot determine whether the current client is using a proxy server or not, they will let it through.

There are a huge number of available public proxy servers on the Internet. However, it is worth noting that using freely available and public proxy servers is not a safe solution. Also, due to the fact that the proxy server data is public, there is a huge chance that several Internet users will use the same proxy server. This option will heavily load the proxy server, thereby reducing the speed of sending and transmitting content through the web server. On the other hand, there are paid solutions that will allow us to use private proxy servers. Since the goal of our study is to build an algorithm and show the work and capabilities of our parser, and not to create a fully finished product, I will use free public proxy servers. For this program, I will use the "Node.js Proxy-list" library, which contains a list of available proxy lists [20]. This list is constantly updated, which indicates that this solution is a good one.

The Selenium tool has the ability to add and configure proxy servers at the root of the project itself [38]. This feature allows you to work directly with the list of proxies. You can also add a randomized selection from the proxy list.

Bypass the "Convert text to image" blocking.

Bypassing this blocking can be done using special optical character recognition programs. These programs exist in Node.js libraries, such as the Node-Tesseract library [13].

It should be noted that the server usually precisely changes user data or phone number data by converting text into an image. Due to the fact that parsing phone numbers and other sensitive data is unethical, this type of data will not be parsed. That is why we should assume that we will not encounter this type of blocking.

Bypass blocking "Data spoofing"

Bypassing this lock using randomized algorithms or using the Selenium tool is not possible. One of the main reasons is the logic of solving this problem. The program cannot determine whether the parsed data is false or true. To fully check this, all that needs to be done is to parse all of the data from the site and then do any other analysis.

On the other hand, you can protect yourself by changing the randomized user-agent or changing the proxy server. Since the data in the browser is always changing, the server won't need to replace the real data.

Bypass blocking "Changing the site structure"

Changing the structure of a site can completely change the architecture and class names. The solution to this problem is a complete change in the site parsing structure, i.e., it is necessary to analyze what has changed and change the class names to the current data. Bypassing this block is a relatively easy task. However, despite the ease of decision, site programmers can change the structure of the site at any time. That is why it is necessary to constantly monitor the relevance of the parser program. The Selenium tool will throw an error when the site structure is changed, saying that the site structure has been changed.

After looking at lock traversal and parsing feature traversals, we can say that Selenium tools and randomized algorithms are the most common solutions among them all. It should be noted that these tools do not solve all lock bypasses. However, it should be assumed that the use of these tools will significantly help achieve our goal.

3.5 Choose a website with strong protection against parsing.

The parser is always made for a specific site. The structure or functionality of the parser may be similar, but it is impossible to make a completely universal parser for all sites in the world. Before creating a parser, it is necessary to analyze several large marketplaces and see whose protection is stronger. Since the purpose of our dissertation is to create a parser for bypassing protections using randomized algorithms, we need to take several large web services and analyze their protections. For analysis, let's take the two largest marketplaces from two countries: Market.kz from Kazakhstan and Avito.ru from Russia. To parse data, we will create a small program using the Node.js library node-osmosis [7]. This script belongs to the script type parser [24]. This means that it works directly with the server's HTML content using GET and POST requests. Using the script

type of the parser will allow you to find out how fast the parsing protection works on the web server. Also, this type of parser is much faster than browser-based parsers [24].

3.5.1 Project initialization on Node-osmosis

To initialize a Node.js project, you need to open a command prompt and type the "npm init -y" command. Next, we create an index.js file, in which we will carry out all the necessary operations.

```
temirlankudabayev@MacBook-Pro-Temirlan thesis % npm install osmosis --save-dev
npm WARN idealTree Removing dependencies.osmosis in favor of devDependencies.osmosis

up to date, audited 242 packages in 5s

20 packages are looking for funding
  run `npm fund` for details

5 high severity vulnerabilities

To address issues that do not require attention, run:
  npm audit fix

To address all issues (including breaking changes), run:
  npm audit fix --force
```

Figure 3.14: Installing the osmosis library

Then you need to install the Node-Osmosis library using the "npm install osmosis –save-dev" command as shown in Figure 3.14. After installation, you need to write a codebase that will be common to compare the protection of two sites.

```
(async () => {
    console.time( label: 'starting');
    const { results, errorCounter } = await getParsedPages( {url, pageNumber}: {
        url: BASE_URL,
        pageNumber: MAX_PARSED_PAGE
    });

    console.timeEnd( label: 'starting');
    console.log('total number of info: ', results.length);
    console.log('total number of failed tasks: ', errorCounter);
```

Figure 3.15: Common codebase for security comparison

Figure 3.15 shows the overall code base for comparing the protection of the Market.kz and Avito.ru platforms. As shown in Figure 3.15, the code contains a generic asynchronous getParsedPages function that initializes the parser. This function also takes URL and PageNumber arguments, where URL is the base URL of the page and PageNumber is the maximum number of pages to be parsed.

To compare the results, we will take such parameters as the total time of the parser, the number of successfully parsed data and the number of failed tasks, as shown in Figure 3.15. It is worth noting that the number of failed tasks will not always mean good protection. If there are problems with the Internet, then, accordingly, the number of faked tasks will be greater.

3.5.2 Parsing protection overview for Market.kz

To parse Market.kz, you need to select any section that is suitable for testing. In this test, we will select the "Services" section, take its URL and set the URL variable, and set the PageNumber value to 50.

In order for the browser to understand what data we need to collect for the test, it is necessary to analyze the basic advertisement card of the Market.kz portal.

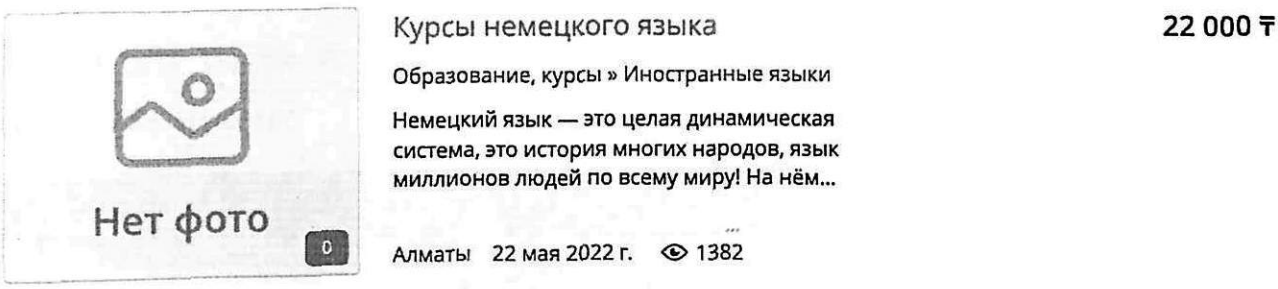

Figure 3.16: Market.kz standard advertisement card

The basic structure of the card consists of the following data: advertisement photos, title, price, category, mini description, city, date, and number of views as shown in Figure 3.16. For the test, you need to choose three parameters. In this case, take the parameters: name, city, and price.

```
const getPage = async currentUrl => {
    return new Promise( executor: (resolve, reject) => {
        osmosis
            .get(currentUrl)
            .set({
                title: ['.ddl_product.a-card .a-card__link'],
                city: ['.ddl_product.a-card .card-stats__item:first-child'],
                price: ['.ddl_product.a-card .a-card__price']
            })
            .data(function({ title, city, price }) {
```

Figure 3.17: Card data parsing.

Figure 3.17 shows the node-osmosis library in action. The osmosis function receives a URL from where data on cards regarding CSS classes should be taken. Furthermore, the program saves these results in an array for further analysis.

```
    price: '45 000 ₸'
  }
]
starting: 1:40.439 (m:ss.mmm)
total number of info:  861
total number of failed tasks:  9
```

Figure 3.18: Market.kz parsing results

Figure 3.18 shows the results of parsing data from the Market.kz website. A total of fifty pages were saved in a minute and forty seconds. There were nine faked tasks in total.

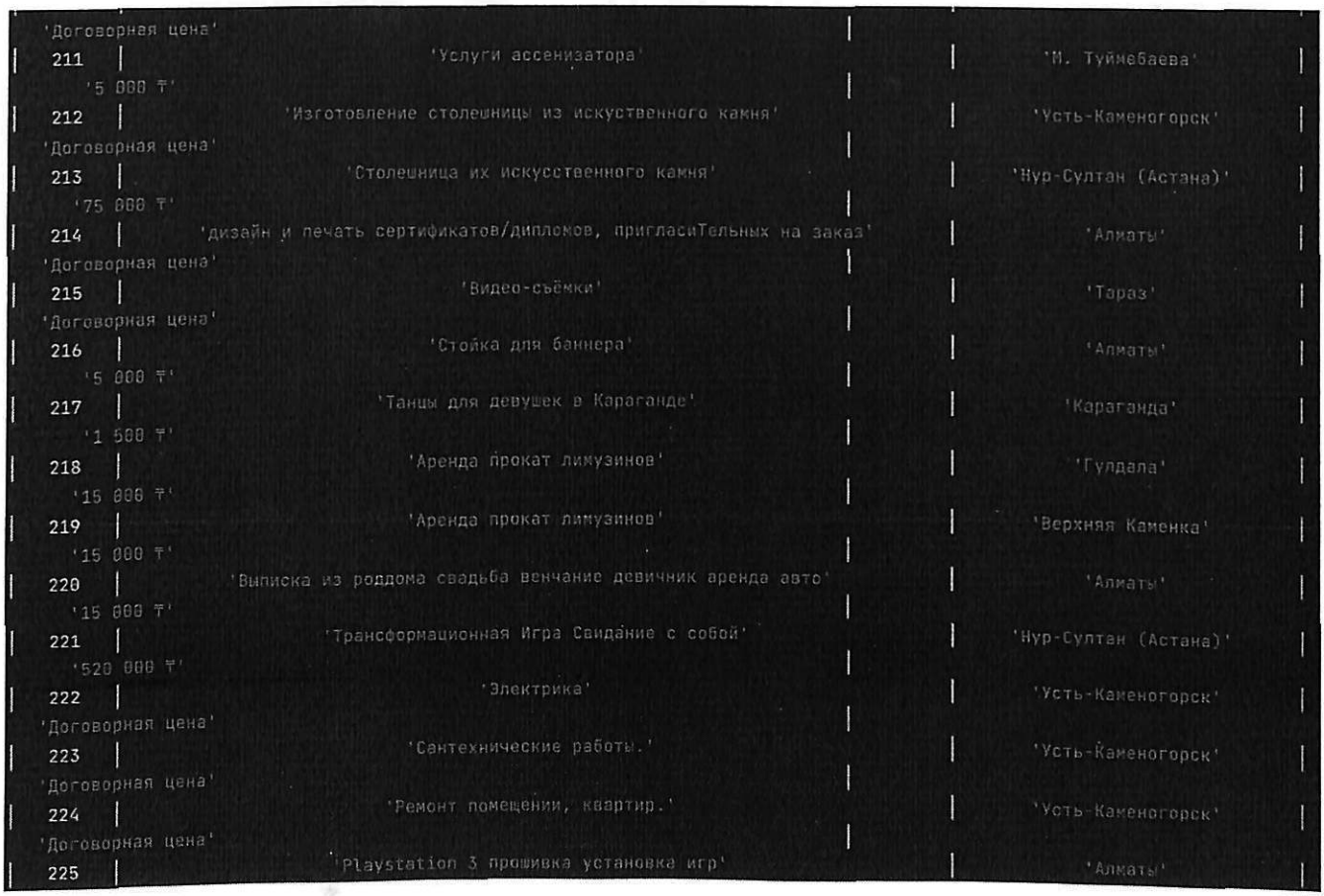

Figure 3.19: Parsed Market.kz data

Figure 3.19 shows part of the parsed data from the Market.kz marketplace. All of the above factors indicate that despite nine faked tasks, script parsers can quickly parse the necessary and necessary data from this marketplace. This is to say that the protection of this site is bypassed.

3.5.3 Parsing protection overview for Avito.ru

To parse Avito, you must also select any available section. In this test, we will select the section "houses in Russia", take its URL and set the value of the variable URL. We will also set the PageNumber value to 50. We will also analyze the basic Avito advertisement card below.

Figure 3.20: Avito basic card structure.

Figure 3.20 shows the structure of the Avito marketplace base card. The card consists of a picture, an advertisement name, a price, and an additional description. For data parsing, we will take only two parameters: name and price.

```
const getPage = async currentUrl => {
    return new Promise( executor: (resolve, reject) => {
        osmosis
            .get(currentUrl)
            .set({
                title: ['.iva-item-root-_lk9K.photo-slider-slider-S15A_ .title-root-zZCwT'],
                price: ['.iva-item-root-_lk9K.photo-slider-slider-S15A_ .price-text-_YGDY']
            })
            .data(function({ title, city, price }) {
```

Figure 3.21: Parsing Avito card data.

Figure 3.21 shows the parsing of card data using the Osmosis library. Figure 3.17 and Figure 3.21 differ only in different CSS class elements due to the different structure of the Avito and Market sites.

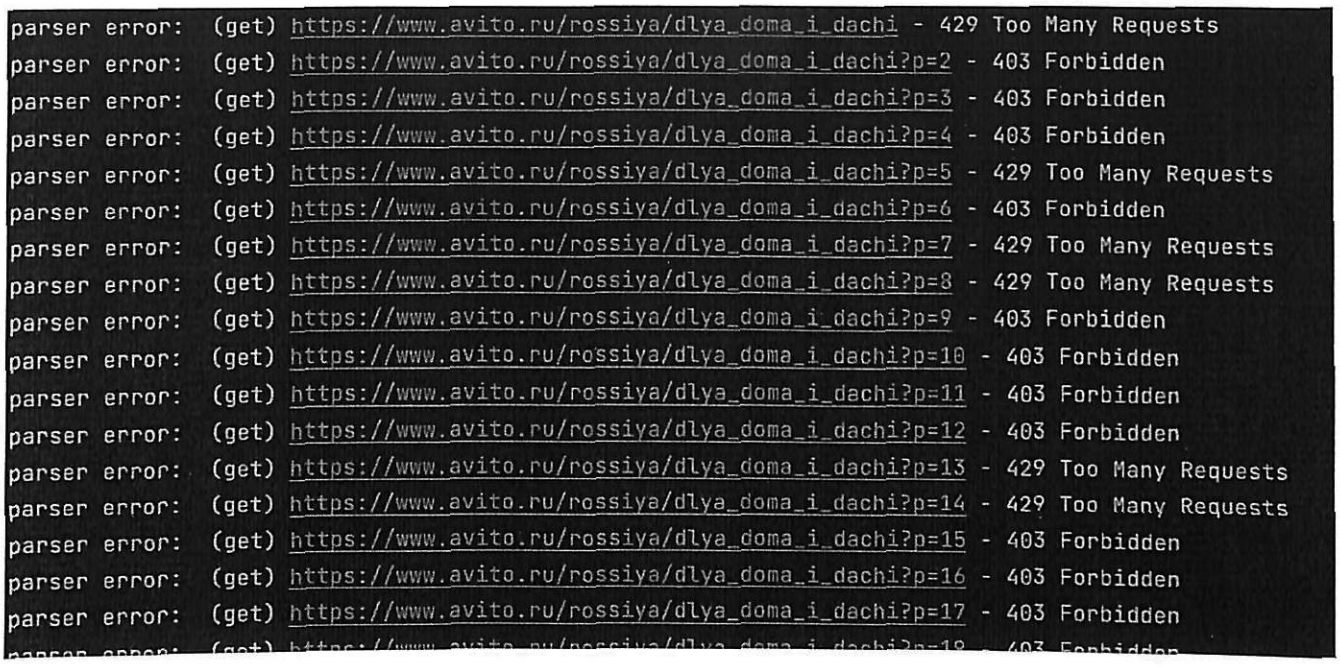

Figure 3.22: Data parsing errors with Avito.

After starting the parser, instead of parsing logs, we see 403 Forbidden and 429 Too Many Requests errors, which are shown in Figure 3.22. A 403 Forbidden server error means that the content on the page you are trying to load is restricted or unavailable [8]. Too Many Requests indicates that the user sent too many requests per unit of time [9]. The response returned by the server contains an explanation and may also include a Retry-After header. This header indicates the amount of time to wait before retrying the request.

The result shows that when trying to parse using the Node-Osmosis library of the Avito marketplace, the protection does not let the parser inside the service. This indicates a basic check for JavaScript rendering on the client as well as checking for browser fingerprints. Restarting the parser repeatedly results in the same errors as shown in Figure 3.22.

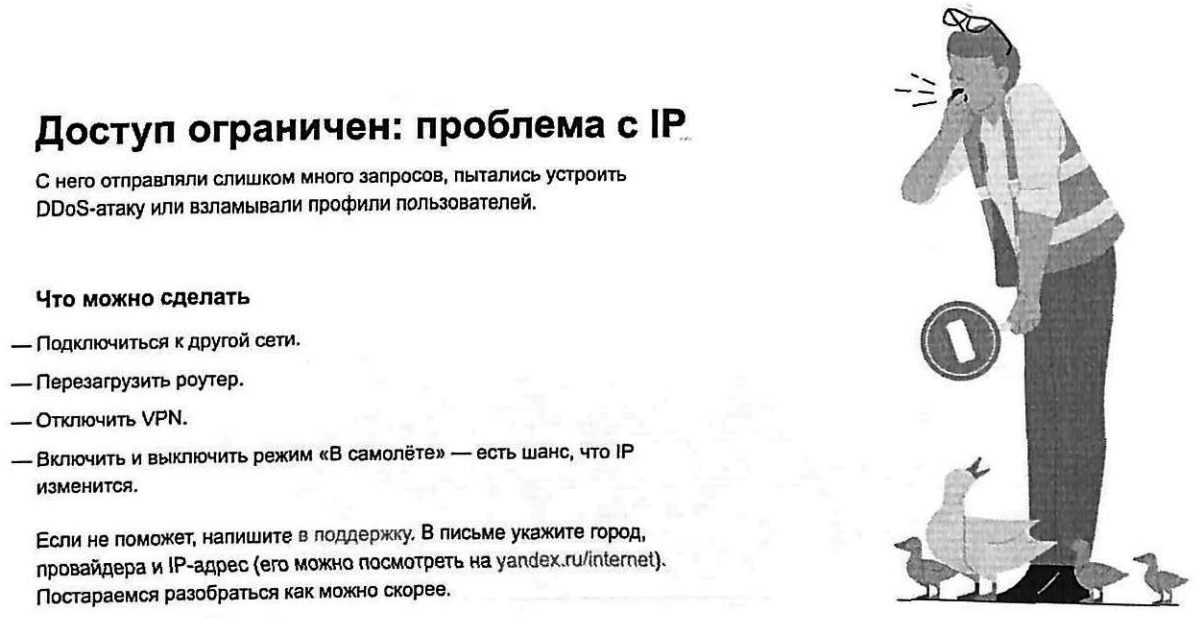

Figure 3.23: User interface example after blocking by IP.

After trying to parse the Avito website using the Node-Osmosis script parser, it was discovered that the service banned us by IP address. This means that after the protection detects a parsing attempt, it automatically reads the IP address of the server and adds it to the blacklist. Figure 3.23 shows what the page of a user banned by IP looks like.

All of the above factors indicate that the Avito marketplace has good protection. Unlike the Marketplace, the Avito service is banned by IP address. It is likely that the marketplace could have changed the ad data. However, unlike Avito, Market allowed the bot to be on the website. These results mean that the Avito service will be selected for the study.

3.6 Writing a program in Node.js using the Selenium library

To initialize the program, you need to create a new Node.js project. As in step 2.3.1, you need to open a command prompt and enter the command "npm init -y". Next, you need to create the main index.js file, where we will write the main code of our program.

```
temirlankudabayev@MacBook-Pro-Temirlan thesis % npm i selenium-webdriver chromedriver

changed 1 package, and audited 242 packages in 4s

20 packages are looking for funding
  run `npm fund` for details

5 high severity vulnerabilities

To address issues that do not require attention, run:
  npm audit fix

To address all issues (including breaking changes), run:
  npm audit fix --force

Run `npm audit` for details.
```

Figure 3.24: Installing chromedriver / selenium-webdriver.

Next, you need to install a tool for parsing data from a web page. Figure 3.24 shows the installation of chromedriver and selenium-webdriver. After installing these packages, we need to import them and use them in our codebase.

```
(async () => {
    try {
        // Этап 1. Инициализация самого драйвера
        const driver = await getSeleniumDriver();

        // Этап 2. Использование драйвера в самом парсинге
        const results = await getParsingResults(driver);

        // Этап 3. Сохранение и вывод результатов в Excel
        await saveResults(results);
    } catch (error) {
        commonExceptionHandler(error);
        throw error;
    }
})();
```

Figure 3.25: The main structure of the parser function.

Figure 3.25 shows the main function of the parser. This function consists of three stages: Selenium driver initialization, data parsing, and results processing.

```
const getSeleniumDriver = async () => {
    const chromeOptions = new chrome.Options();
    chromeOptions.addArguments('headless');

    return new webdriver.Builder()
        .forBrowser( name: "chrome") !Builder
        .setChromeOptions(chromeOptions) !Builder
        .build();
}
```

Figure 3.26: getSeleniumDriver function.

Figure 3.26 shows the code for the first step in the parsing process. This

getSeleniumDriver function initializes the chrome browser itself in headless mode. After completing the settings, it builds the configuration itself and returns a completely standalone browser.

```
const getParsingResults = async (driver) => {
    const results = [];
    let errorCount = 0;

    try {
        const pageList = getPageList( totalPageNumber: 20);
        await asyncForEach(pageList,  callback: async currentPage => {
            const currentURL = currentPage > 1 ? `${PAGE_URL}?&p=${currentPage}` : PAGE_URL;
            console.log('Переход на страницу: ', currentURL);

            const response = await getSinglePage(driver, currentURL);
            console.log(`Страница - ${currentPage}: Собрано`, response.length);
            results.push(response);

            console.log('Ожидание...');
        });

        return results;
    } catch (error) {
        errorCount++;
```

Figure 3.27: getParsingResults function.

Figure 3.27 shows the code for the getParsingResults function. This function is the heart of this program because it is where the data is parsed. Before we start parsing, we initialize the number of pages to be parsed. Next, we alternately create a special link to which our parser should go. After the transition, the parser parses all the necessary data from the page using the getSinglePage function. It then stores the parsed data into the results array. If a parser error occurs, a system crash or a problem with access, then we display the program logs and count the number of errors using the errorCount variable.

```
const getSinglePage = async (driver, currentUrl) => {
    const results = [];

    try {
        await driver.get(currentUrl);
        const elements = await driver.findElements(webdriver.By.className( name: 'iva-item-root-_lk9K'));

        for (let element of elements) {
            // Получение элементов
            const title       = await element.findElement(webdriver.By.className( name: 'title-root-zZCwT'));
            const information = await element.findElement(webdriver.By.className( name: 'iva-item-text-Ge6dR'));
            const price       = await element.findElement(webdriver.By.className( name: 'price-text-_YGDY'));
            const geolocation = await element.findElement(webdriver.By.className( name: 'geo-address-fhHd0'));

            // Получение текстов
            const titleText       = await title.getText();
            const informationText = await information.getText();
            const priceText       = await price.getText();
            const geolocationText = await geolocation.getText();

            // Работа с текстом
            const [nameText, ageText] = titleText.split(',');
            const [mileageText, modificationText, bodyText, driveUnitText, engineText] = informationText.split(',');
            const engineVolumeText = modificationText.trim().substring(0, 2);
```

Figure 3.28: getSinglePage function.

Figure 3.28 shows the first part of the code for the getSinglePage function. This function goes to the current page of the site. Next, she collects all the cards of the current ad. The next step, she goes to each card individually and collects all the necessary information: title, information, price, geolocation. Next, we group all the necessary information and work with the text.

```
// Запись результата
const result = [
    ['name', nameText],
    ['age', ageText],
    ['mileage', mileageText],
    ['modification', modificationText],
    ['body', bodyText],
    ['driveUnit', driveUnitText],
    ['engine', engineText],
    ['engineVolume', engineVolumeText],
    ['price', priceText],
    ['geolocation', geolocationText],
].reduce((previousValue :{} , currentValue :(string|any)[] ) => {
    const [keyName, keyValue] = currentValue;
    previousValue[keyName] = keyValue.trim();

    return previousValue;
}, {});

results.push(result);
```

Figure 3.29: getSinglePage function.

Figure 3.29 shows how the parsed results of parsed declarations are handled inside the getSinglePage function. We remove all extra lines of each value using the trim function. Next, we store all the results in the results array, and after the end of the data parsing, we return this array.

```
const saveResults = async (results) =>
    new Promise( executor: async (resolve, reject) => {
        try {
            const schema = [
                {
                    column: 'Название',
                    type: String,
                    value: result => result.name
                },
                {column: 'Год выпуска'...},
                {column: 'Пробег'...},
                {column: 'Модификация'...},
                {column: 'Привод'...},
                {column: 'Тип кузова'...},
                {column: 'Тип двигателя'...},
                {column: 'Объём двигателя'...},
                {column: 'Цена'...},
                {column: 'Местоположение'...}
            ]

            await writeXlsxFile(results, {
                schema,
                filePath: './output_file.xlsx'
            });
        } catch (error) {
            reject(error);
        }
    });
```

Figure 3.30: saveResults function.

Figure 3.30 shows the code for the saveResults function. This function saves the parsed data to Excel using the write-excel-file library [22]. To set up saving data, you need to pass the results values to the writeXlsxFile function, as well as an object with the schema and filePath settings.

Название	Год выпус	Пробег	Модифика	Тип кузова	Привод	Тип двига	Объём дв	Цена	Местопол
Hyundai Ac	2006	165 000 км	1.5 MT (92	седан	передний	бензин	1.5	245 000 ₽	Краснодар
Kia Rio	2015	181 300 км	1.4 MT (10	седан	передний	бензин	1.4	390 000 ₽	Москва
Toyota Mai	1993	100 000 км	3.0 AT (220	седан	задний	бензин	3.0	449 999 ₽	Москва
Toyota RAV	2019	7 000 км	2.5 AT (199	внедорож	полный	бензин	2.5	1 630 000 ₽	Вологодск
Nissan X-Tr	2013	68 000 км	2.0 CVT (14	внедорож	полный	бензин	2.0	1 399 000 ₽	Санкт-Пете
Opel Insign	2009	265 000 км	2.0 AT (220	седан	передний	бензин	2.0	520 000 ₽	Краснодар
Mitsubishi	2008	159 900 км	1.6 AT (98	седан	передний	бензин	1.6	335 000 ₽	Нижегород
Kia Rio	2014	183 000 км	1.4 MT (10	седан	передний	бензин	1.4	320 000 ₽	Саратовск
Hyundai So	2014	Битый 143 000 км	1.6 AT (123	седан	передний	бензин	1.6	415 000 ₽	Москва
ВАЗ 2114 S	2007	180 000 км	1.6 MT (81	хетчбэк	передний	бензин	1.6	143 000 ₽	Свердловс
Audi A6	2012	130 000 км	2.0 CVT (18	седан	передний	бензин	2.0	870 000 ₽	Белгородс
Honda Civi	2008	152 000 км	1.8 AMT (1	хетчбэк	передний	бензин	1.8	399 000 ₽	Санкт-Пете
Hyundai So	2013	107 000 км	1.6 AT (123	седан	передний	бензин	1.6	600 000 ₽	Республик
LADA Prior	2007	236 453 км	1.6 MT (98	седан	передний	бензин	1.6	160 000 ₽	Краснодар
SEAT Toled	1992	285 000 км	2.0 AT (115	хетчбэк	передний	бензин	2.0	100 000 ₽	Калинингр
Hyundai Ac	2008	156 899 км	1.6 MT (11	седан	передний	бензин	1.6	313 000 ₽	Ставропол
Peugeot 40	2011	123 000 км	2.4 CVT (17	внедорож	полный	бензин	2.4	620 000 ₽	Москва
ВАЗ (LADA)	2007	255 000 км	1.6 MT (98	седан	передний	бензин	1.6	98 000 ₽	Ивановска
Kia Ceed	2011	191 000 км	1.6 AT (122	хетчбэк	передний	бензин	1.6	575 000 ₽	Краснодар
BMW 3 сер	2002	24 500 км	2.0 MT (14	седан	задний	бензин	2.0	570 000 ₽	Саратовск
Mazda 3	2006	180 000 км	2.0 MT (15	седан	передний	бензин	2.0	346 000 ₽	Белгородс

Figure 3.31: File Output *file.xlsx*.

Figure 3.31 shows an example of saved data in Excel. The parsed data looks very structured and clean. After finishing the implementation of our basic algorithm, we need to check the stability of our program with the command: "node selenium_avito.js". If various errors or bugs come out, we must fix them. After solving all the problems, we move on to the next stage.

3.7 Implement randomized algorithms in the program base.

The server-side Node.js language has a large number of internal libraries, classes, and imported APIs available. The global class Math is one of these. The Math object is a built-in object that stores various mathematical constants and functions in its properties and methods. The Math object is not a function object [10].

```
console.log('Check type of Math class: ', typeof Math);

const randomNumber = Math.random();
console.log('Random number is: ', randomNumber);
```

Figure 3.32: The Math class in Node.js.

Figure 3.32 shows the Math global class in action in a Node.js program. In addition to various built-in functions, the Math class has a special random function. This is a function that returns random numbers.

```
temirlankudabayev@MacBook-Pro-Temirlan thesis % node selenium
Check type of Math class:  object
Random number is:  0.26621206142769793
temirlankudabayev@MacBook-Pro-Temirlan thesis % node selenium
Check type of Math class:  object
Random number is:  0.8867865451126491
temirlankudabayev@MacBook-Pro-Temirlan thesis % node selenium
Check type of Math class:  object
Random number is:  0.8333343977563437
```

Figure 3.33: Run a script to show the result.

Figure 3.33 shows the math class in action. In this case, we ran the script three times, and all three times the Math.random() function returned different results. The range of returned numbers is represented by values from 0 (including 0, that is, it can return 0) to 1 (not including 1, that is, it cannot return one). Random algorithms will be put into action with the help of the Math.random() function.

Several of the things we talked about above will require random algorithms for our parser:

- Randomized wait function
- Function to get a random proxy server

- Function to get a random User-Agent parameter

3.7.1 Implementation of the wait function with randomized algorithms

To create a randomizedWaiter function, you need to create a getRandomNumberBetween helper function. This helper function will take two parameters, min and max, where min is the minimum value and max is the maximum value, respectively. The getRandomNumberBetween function will return a random number using the Math.random() function, from the min value to the max value, inclusive. Using the value of the getRandomNumberBetween function, we will set the randomized time. The setTimeout function built into Node.js will help us do this operation. Note that the wait function is an asynchronous function.

```
// Функция для рандомизированного ожидания асинхронный функции
const MIN_WAITER = 4;
const MAX_WAITER = 10;

const getRandomNumberBetween = (min, max) => {
    return Math.floor( x: Math.random() * (max - min + 1) + min)
}

const randomizedWaiter = () =>
    new Promise( executor: (resolve) => {
        const randomTime = getRandomNumberBetween(MIN_WAITER, MAX_WAITER);
        setTimeout( handler: () => resolve,  timeout: randomTime * 1000);
    });

module.exports = {
```

Figure 3.34: randomiedWaiter function

Figure 3.34 shows the code for the above snippets of the function. We set the minimum and maximum waiter values at the very beginning. By default, we set minimum waiter value to four and maximum waiter value to ten.

```
    const structuredPageData = await getParsedPage(pageNumber);
    parsedResults.push([...parsedResults, structuredPageData]);

    // Ожидаем небольшой количество времени чтобы сервер не загружать сервер
    await randomizedWaiter();
} catch (error) {
    commonExceptionHandler(error);
```

Figure 3.35: Using the asynchronous randomizedWaiter function.

Figure 3.35 shows the use of the randomized asynchronous function randomizedWaiter in the main parsing algorithm. The ideal application of this function is to insert this function at the moment the parsing of a certain page is completed. In practice, it will happen like this: The parser goes to a specific page that needs to be parsed. Furthermore, the parser will extract the necessary information using a special Selenium Webdriver API. After all the necessary information has been collected, we must save it. After saving the results, we will call the randomizedWaiter function to simulate waiting.

3.7.2 Function to get a random proxy server

The principle of this function is that the function receives a random proxy server from the general list of available public proxy servers and passes this list to the main parser algorithm. The getRandomNumberBetween helper function, which is used in paragraph 3.7.1, is also useful for this function. To get a list of proxy servers, you need the "Node.js proxy-lists" library [20]. To install it, you need to open the project terminal and type the command "npm install proxy-lists –save-dev".

```
temirlankudabayev@MacBook-Pro-Temirlan thesis % npm install proxy-lists --save-dev
npm WARN idealTree Removing dependencies.proxy-lists in favor of devDependencies.proxy-lists

up to date, audited 242 packages in 2s

20 packages are looking for funding
  run `npm fund` for details

5 high severity vulnerabilities

To address issues that do not require attention, run:
  npm audit fix

To address all issues (including breaking changes), run:
  npm audit fix --force

Run `npm audit` for details.
```

Figure 3.36: Installing the proxy-lists library.

Figure 3.36 shows the result of installing the proxy-lists library. After installing this library, you need to import it into the code base of our project. This can be done using the required keyword.

```
const ProxyLists = require('proxy-lists');

const {commonExceptionHandler} = require("./utils/requests");

const getProxyList = (protocols :any[]  = []) => {
    return new Promise( executor: resolve => {
        let proxies = [];
        ProxyLists.getProxies( options: { protocols })
            .on('data', p => p.forEach(p => proxies.push(`${p.ipAddress}:${p.port}`)))
            .once('end', () => resolve(proxies));
    })
};
```

Figure 3.37: getProxyList function.

Figure 3.37 shows the getProxyList function, which is responsible for getting a complete list of all available proxy servers. After importing the proxy-lists library, we use the getProxies method of this class to create a request to get all available proxy servers. In response, we receive a list of objects of type key: value. In this case, we have several keys: ipAddress and port, and their values, respectively. To

	HTTP	HTTPS	SOCKS
Fast download speed			
Secure connection support			
Completely anonymous protocol			

Table 3.1: Proxy Protocol Comparison

standardize the response, it is necessary to translate these values into a single string consisting of the value of ipAddress and port. As a final result, we get a list of proxy servers from which we can select the proxy server we need. It is worth noting that the ProxyLists.getProxies() method accepts the protocols value. This value is an array of proxy protocols that we need.

To improve the understanding of the code, it is necessary to consider what the proxy server protocol is. Depending on the protocol used by the proxy server, they are divided into many types, but the most popular are HTTP, HTTPS, or SOCKS [34].

An HTTP proxy is the most common type of proxy. The main purpose is to organize the work of browsers and other programs using the TCP protocol. How it works: a program or browser sends a request to a proxy server to open a specific resource URL. The proxy server gets information from the resource that was asked for and sends it to your browser.

In fact, this is an HTTP proxy; the letter "S" in this case means "secure" (protected) with support for a secure SSL connection. These proxies are used when sensitive information needs to be transmitted.

To date, Socks proxy is the most advanced information transfer protocol. The Socks proxy does not transmit information about your IP address. The website will not be able to detect the use of a proxy. The connection to the website will be absolutely transparent, just as if you were working with it directly. In this case, the website will see the IP address of the proxy and not your real IP address.

Table 4.1 shows a comparison of proxy protocols regarding the parameters: Web page loading speed, secure connection support, and fully anonymous protocol [34]. If this proxy protocol supports a comparable parameter, then the table cell of this proxy will be colored green, otherwise it will be red. Based on the results of Table 1, it should be noted that using the Socks protocol is a better solution than using the HTTP or HTTPS protocols. In this case, we will use the Socks protocol for our algorithm.

```
const getRandomProxy = async (protocols : any[]  = []) => {
    try {
        const availableProxies = await getProxyList(protocols);

        if (Array.isArray(availableProxies) && availableProxies.length) {
            const randomIndex = getRandomNumberBetween(0, availableProxies.length - 1);
            return availableProxies[randomIndex];
        }

        return null;
    } catch (error) {
        commonExceptionHandler(error);
    }
}
```

Figure 3.38: getRandomProxy function.

Figure 3.38 shows the code for the getRandomProxy function. This function receives a list of protocols and returns a random proxy server. To begin with, a request is sent to get all available proxy servers. Next comes the check for the existence of this array. If the array exists and the list of proxy servers is not empty, then we continue to work with proxy servers, otherwise we return null. After we have made sure that we have a list of proxy servers available, we must get any of the available proxy servers. This can be done using the getRandomNumberBetween helper function, which will return the randomized array index to us. Next, using this index, we return the selected proxy server.

```
const ALLOWED_PROXY_PROTOCOL = 'https';

const getSeleniumDriver = async () => {
    const chromeOptions = new chrome.Options();
    chromeOptions.addArguments('headless');

    const proxyAddress = await getRandomProxy(ALLOWED_PROXY_PROTOCOL);

    if (proxyAddress) {
        chromeOptions.addArguments(`--proxy-server=${ALLOWED_PROXY_PROTOCOL}://${proxyAddress}`);
    }
```

Figure 3.39: using the getRandomProxy function.

Figure 3.39 shows the code for the getSeleniumDriver function, which returns

the browser's driver with all the necessary settings. Inside the getSeleniumDriver function, we use the getRandomProxy function described above and wait for its result to be executed. If we find any proxy server, then we use it as an argument to our driver using the addArguments function.

...

3.7.3 Function to get a random User-Agent parameter

Selenium Webdriver already has the ability to change the browser for its intended purpose, but this process takes quite a long time and requires the installation of additional huge packages. Writing a function that will return a random User-Agent parameter is another way to solve this problem.

To create this function, you must have the User-Agent parameter data ready. To do this, download the JSON file useragent-data.json [35]. This file is made up of an array of 822 objects, each of which stores information about the User-Agent.

```
// Функция для получения рандомизированного user-agent-а из списка
const getRandomUserAgent = () => {
    if (Array.isArray(userAgentsList) && userAgentsList.length) {
        const randomIndex = getRandomNumberBetween(0, userAgentsList.length - 1);
        return userAgentsList[randomIndex];
    }

    return null;
}
```

Figure 3.40: getRandomUserAgent function.

Figure 3.40 shows the final code for the getRandomUserAgent function. This function checks for the presence of the User-Agent array. By using the getRandomNumber function, it gets a randomized index that the function will use to return a specific User-Agent.

```
const userAgent = getRandomUserAgent();
const userAgentArgument = `user-agent="${userAgent}"`;
chromeOptions.addArguments([userAgentArgument]);

return new webdriver.Builder()
    .forBrowser( name: "chrome") !Builder
    .setChromeOptions(chromeOptions) !Builder
```

Figure 3.41: Using the getRandomUserAgent function.

Figure 3.41 shows the use of this getRandomUserAgent function inside the getSeleniumDriver function. After receiving a randomized User-Agent parameter, we add it as an argument to our parser settings using the addArguments function.

3.8 Test the work of the parser with randomized algorithms and without randomized algorithms.

We have covered the theory behind the website security system, looked at the four most popular features for detecting a parser, as well as four ways to block parsers. Next, we studied the theory regarding randomized algorithms and realized that randomness is used in many areas of science and technology. Also, for our study, we analyzed two popular web services, avito.ru and market.kz, and assessed their security systems. In the next step, we used the Selenium tool to write our parser program and added random algorithms to their code.

Before proceeding to the results, it is worth considering the moment of testing the operation of the program itself. There are no programs without errors: any program can give unpredictable results in response to the most common actions. The developer, most likely, will not notice these defects in the code, but they can poison the life of the end user. There are errors, small and insignificant, and there are also such that everything stops working.

After the testing is complete, you can proceed to the result section. We will test the work of two programs: Parser on Selenium without randomized algorithms and Parser on Selenium with randomized algorithms. In the next section, we will

talk about the results of these tests and do a full analysis of different program cases.

4. Data and Results

In this section, we will test the stability of the parser with randomized algorithms. To compare the results, we will take a copy of the written parser but without randomized algorithms. A comparison of results will take place with respect to these parameters.

- Total number of pages parsed
- Total amount of data parsed
- The text in the entries may be of any length.
- Data parsing speed per unit of time.

Page category	Page subcategory	The number of advertisements	The number of pages
Hyundai	Accent	2874	58
Hyundai	Santa Fe	3016	61
Hyundai	Tucson	2918	59
Hyundai	Sonata	3174	64
Hyundai	Solaris	18735	100

Table 4.1: Initial data for parsing

Table 4.1 shows the initial data that the parsers will process. In this case, for the experiment, we will take the category of cars, and among them we will select the subcategory of Hyundai. The Hyundai car brand has a huge number of models. Among them, the five most popular were chosen for the experiment:

Accent, Santa Fe, Tuscon, Sonata, Solaris, with 2874, 3016, 2918, 3174, and 18375 ads, respectively. Before starting the experiment, you need to know that Avito has a limit of one hundred pages per category. This limitation means that, despite the large number of declarations, any parser will not be able to go beyond the hundredth page.

Depending on each parameter, in order to visualize the data, we will build tables. For visual distinction, the parser with the best performance on this parameter will be green, otherwise red. If the indicators of this parameter are equal to each other, then both parsers will be green. For convenience in data naming, a parser with randomized algorithms will be called Parser 1, and a parser without using randomized algorithms will be named Parser 2.

	Number of pages	Parser #1	Parser #2
Hyundai Accent	58	15	58
Hyundai Santa Fe	61	20	61
Hyundai Tucson	59	19	59
Hyundai Sonata	64	20	62
Hyundai Solaris	100	21	99

Table 4.2: Number of parsed pages relative to the parser.

Table 4.2 shows the number of parsed pages relative to the parser. The results show that the parser with randomized algorithms parsed almost all the required pages. This program has fully worked out sections of the Hyundai Accent, Hyundai Santa Fe, and Hyundai Tucson. The inaccuracies occurred in the Hyundai Sonata and Hyundai Solaris pages, where 62 out of 64 pages and 99 out of 100 pages were parsed, respectively. Parser 2, unlike parser 1, parsed a much smaller number of pages. The average number of parsed parser pages without the implementation of randomized algorithms is nineteen pages. This happens as a result of blocking the parser, the results of which we will consider below.

	Parser #1	Parser #2
Hyundai Accent	836	3123
Hyundai Santa Fe	1120	3374
Hyundai Tucson	1064	3272
Hyundai Sonata	1118	3469
Hyundai Solaris	1171	5574

Table 4.3: Number of parsed data relative to the parser.

Table 4.3 shows the amount of parsed data relative to the parser. Table 4.2 and Table 4.3 are interdependent. Since Parser 2 parsed many more pages, it received a much larger amount of data accordingly. The results show that the parser with randomized algorithms parsed only 18812 rows of data, while the parser without using randomized algorithms parsed 5309 rows of data. There are 13503 lines of data difference between these two datasets.

	Number of mistakes	Number of locks
Parser #1	3	0
Parser #2	0	5

Table 4.4: Total number of errors and locks relative to the parser.

Table 4.4 shows the results of the parsed data table against the parser. The column number of errors indicates the number of non-critical errors that occurred during the program's operation. The number of blocks column shows how many times the website protection found that the program is a parser and blocked access to the web resource.

These results show that the parser with randomized algorithms bypassed the lock or did not arouse the server's suspicion at all, in contrast to the parser without randomized algorithms with zero and five locks, respectively. On the other hand, a parser with randomized algorithms caused more errors than a parser without randomized algorithms. These errors may be related to factors such as loss of access to the Internet connection, or a very slow Internet connection, or various communication errors from the site. In total, the parser with randomized

algorithms has three errors that came out during the parsing of five large sections of the site. This number of errors is a normal deviation when working with parsers. In order to figure out what errors came out during the blocking of the parser, you need to show the screenshot below.

Figure 4.1: Parser error while parsing data.

Figure 4.1 shows an example of a parser error that was implemented without randomized algorithms. As shown in Figure 4.1, when parsing the Avito website, an error occurred on page 21. This error appears in the getSinglePage function, where the page itself is actually parsed. The main error says "TypeError: Cannot read property 'trim' of undefined." This error points to a line of code where extra data is cleared from lines. The parser did not receive the necessary data to populate the results. There are two options for the event: the ad card has been modified, or the site has blocked the parser. To clarify this error, you need to go to Avito.

Figure 4.2 shows the UI of a client blocked by IP address. We got this result after the error from Figure 4.1 appeared. After going to the Avito website, we receive notifications that we have been blocked. This means that the server

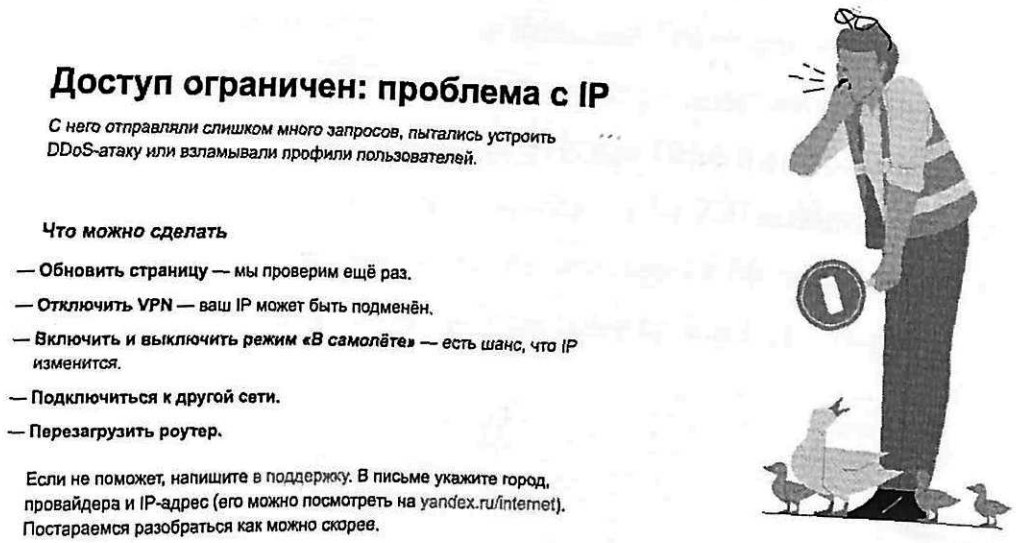

Figure 4.2: Blocking by IP address from the Avito service.

protection has detected suspicious activity and has taken steps to restrict access to the web resource. This figure has a great similarity with Figure 3.23, since then the script parser was also working on the same Avito site. Most likely, the developers of the Avito site managed to slightly change the design of the blocked page because the actions from the Methods Materials section and the current actions take place at different time intervals.

To figure out the average speed of data parsing per unit of time, the average time it took to collect data on one page was used as a measure.

	Parser #1	Parser #2
Hyundai Accent	23.254	15.325
Hyundai Santa Fe	22.556	14.393
Hyundai Tucson	23.439	14.256
Hyundai Sonata	28.102	17.796
Hyundai Solaris	30.872	18.317

Table 4.5: Average parsing time of one page relative to the parser in seconds.

Table 4.5 shows data regarding the average parsing time relative to the parsed section of the site. The data shows that parser 2 was much faster than parser 1. The main reason is the presence of randomized functions such as wait functions, get proxy server functions, and user-agent change functions, which added time to the work of the function. The minimum average time was obtained when working with the Hyundai Tuscon section, with a time of 14.256 seconds. It's important to note that one page of the section loads an average of 56 ads with all the pictures and information needed, so it takes a long time to load one page.

5. Conclusion

The purpose of this work was to develop a Node.js parser using the Selenium tool and implement randomized algorithms to simulate human actions. In this work, we have considered by what popular signs the server protection understands that a website visitor is a parser and what measures the server takes to block the bot. For a qualitative study, the Avito.ru site was chosen, the server protection of which was tested by special fast script parsers. In the results section, we used two different parsers to look at the chosen site, one with random algorithms and one without.

The results showed that the parser on the Selenium tool could parse on average nineteen pages of a marketplace with parsing protection installed before the protection blocked it, in contrast to script parsers, which were blocked on the first attempts to launch the program. The results also showed that the introduction of randomized algorithms into the parser with the Selenium tool to simulate human actions really helps in the stability of the data parsing. The results showed that out of the five parsed categories, the parser with randomized algorithms did not have any locks, unlike the parser without randomized algorithms. On the other hand, the parser with randomized algorithms worked much slower due to the presence of a function with randomized expectation, a function to get proxy servers, and other functions.

In order to improve our research, we may improve the testing performance in the future. In this research, we conducted an analysis regarding some departments at one site. In the future, we may set the goal of parsing all site data as well as running a plurality of such parsers and checking the results of the work. Also, to improve performance, we can set up our own proxy servers or rent available private proxy servers and check the speed of data parsing.

A. Appendix

Year	All Websites	Active Websites
January 2022	1,167,715,133	*198,988,100*
January 2021	1,197,982,359	*199,533,484*
January 2020	1,295,973,827	*189,000,000*
January 2019	1,518,207,412	*182,185,876*
January 2018	1,805,260,010	*171,648,771*
January 2017	1,800,047,111	*172,353,235*
January 2016	906,616,188	*170,258,872*
January 2015	876,812,666	*177,127,427*
January 2014	861,379,152	*180,067,270*
January 2013	629,939,191	*186,821,503*
January 2012	582,716,657	*182,441,983*
January 2011	273,301,445	*101,838,083*
January 2010	206,741,990	*83,456,669*
January 2009	185,497,213	*71,647,887*
January 2008	155,583,825	*68,274,154*

Table A.1: Total Number of Websites by Year

References

[1] Chris Antcliff. *How to add Google reCAPTCHA to a Form.* URL: https://www.bronco.co.uk/our-ideas/how-to-add-google-recaptcha-to-a-form-phphtml. July 31, 2015.

[2] Eugene Bozhenko. *CS:GO Spray Patterns Recoil Compensation.* URL: https://dmarket.com/blog/csgo-spray-patterns/. June 30, 2021.

[3] Yoast BV. *What is site structure and why is it important?* URL: https://yoast.com/site-structure/. November 1, 2018.

[4] CERN. *The birth of the Web.* URL: https://home.web.cern.ch/science/computing/birth-web. 2022.

[5] Kent Chen. *5 OCR Ways to Extract Text from Images on Windows 10.* URL: https://www.nextofwindows.com/5-ocr-ways-to-extract-text-from-images-on-windows-10. June 28, 2018.

[6] Catherine Chipeta. *What is a Proxy Server? A Clear Explanation of How it Works.* URL: https://www.upguard.com/blog/proxy-server. May 12, 2022.

[7] Robbie Chipka. *Github - Node-Osmosis: Web-scrapper for Node.js.* URL: https://github.com/rchipka/node-osmosis. Mar 1, 2019.

[8] Mozilla Corporation. *403 Forbidden - HTTP | MDN.* URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/403. November 26, 2021.

[9] Mozilla Corporation. *429 Too Many Requests - HTTP | MDN.* URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/429. May 13, 2022.

[10] Mozilla Corporation. *Math - JavaScript | MDN.* URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math. February 18, 2022.

[11] Mozilla Corporation. *User-Agent HTTP.* URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/User-Agent. February 18, 2022.

[12] Mozilla Corporation. *What is accessibility? - Learning web development | MDN.* URL: https://developer.mozilla.org/ru/docs/Learn/Accessibility/What_is_accessibility. March 20, 2021.

[13] Carlos Delgado. *Getting started with Optical Character Recognition (OCR) with Tesseract in Node.js.* URL: https://ourcodeworld.com/articles/read/348/getting-started-with-optical-character-recognition-ocr-with-tesseract-in-node-js. January 2, 2017.

[14] Anna Fitzgerald. *What Is the Average Time Spent On a Website?* URL: https://blog.hubspot.com/marketing/chartbeat-website-engagement-data-nj. December 22, 2021.

[15] Yang Guo. *There's Math.random(), and then there's Math.random().* URL: https://v8.dev/blog/math-random. December 17, 2015.

[16] P. Gupta and K. Johari *Implementation of web crawler.* URL: http://troindia.in/journal/ijcesr/vol2iss4/99=101.pdf. 2009.

[17] Neal R Haddaway. *The Use of Web-scraping Software in Searching for Grey Literature.* URL: https://www.researchgate.net/publication/282658358_The_Use_of_Web-scraping_Software_in_Searching_for_Grey_Literature. October 2015.

[18] Erez Hasson. *Bad Bot Report 2021: The Pandemic of the Internet.* URL: https://www.imperva.com/blog/bad-bot-report-2021-the-pandemic-of-the-internet/. Apr 13, 2021.

[19] Avita HelpDesk. *What is number protection?* URL: https://support.avito.ru/articles/1865. 2022.

[20] Charles Hill. *Node.js module and CLI tool to get proxies from publicly available proxy lists.* URL: https://github.com/chill117/proxy-lists. September 6, 2021.

[21] Nick Huss. *How Many Websites Are There in the World?* URL: https://siteefy.com/how-many-websites-are-there/. April 7, 2022.

[22] Nikolay Kuchumov. *Gitlab write-excel-file repository.* URL: https://gitlab.com/catamphetamine/write-excel-file. 2022.

[23] Temirlan Kudabayev. "Calculating the effectiveness of speeding up page loading of the React.js website by implementing the Next.js framework". In: 11 (2022), p. 2.

[24] Temirlan Kudabayev. "Script and browser-based parsers: Efficient type detection based on four parser performance measures". In: 9 (2022), pp. 2–3.

[25] Kaspersky Lab. *What is an IP Address – Definition and Explanation.* URL: https://www.kaspersky.com/resource-center/definitions/what-is-an-ip-address. 2022.

[26] Xurxo Legaspi. *Scraping Dynamic Websites for Economical Data.* URL: https://www.diva-portal.org/smash/get/diva2:1032894/FULLTEXT01.pdf. 2016.

[27] Anticaptcha Development LP. *AntiCaptcha Documentation.* URL: https://anti-captcha.com/apidoc. 2016.

[28] Harsukh Makwana. *Laravel 8 - How to Restrict or Block User Access via IP Address.* URL: https://www.laravelcode.com/post/laravel-8-how-to-restrict-or-block-user-access-via-ip-address. June 2021.

[29] Paula. *What are the methods used against web scraping?* URL: https://www.scraping-bot.io/anti-scraping-methods/. February 14, 2020.

[30] Mina Pêcheuxk. *What is Lorem Ipsum and why you should use it.* URL: https://medium.com/geekculture/what-is-lorem-ipsum-and-why-you-should-use-it-be89f124ff80. August 18, 2021.

[31] Huy Phan. *Building Application Powered by Web Scraping.* URL: https://www.theseus.fi/bitstream/handle/10024/166489/Phan%5C%20Huy%5C%20Thesis%5C%20Building%5C%20Application%5C%20Powered%5C%20by%5C%20Web%5C%20Scraping.pdf?sequence=2. March 1, 2019.

[32] Boris Ryabko. *A Pseudo-Random Generator Whose Output is a Normal Sequence.* URL: https://www.worldscientific.com/doi/abs/10.1142/S0129054121500325?download=true&journalCode=ijfcs. September 22, 2022.

[33] Javier Ochoa Serna. *Design and Implementation of a scraping system for sport news.* URL: https://oa.upm.es/44707/1/PFC_JAVIER_OCHOA_SERNA_2017.pdf. 2017.

[34] Safety service. *Types of Proxy HTTP, HTTPS, Socks.* URL: https://thesafety.us/http-socks-proxy. 2022.

[35] Skratchdot. *GitHub - skratchdot/random-useragent: Get a random user agent.* URL: https://github.com/skratchdot/random-useragent. 2022.

[36] StackOverflow. *Stack Overflow Developer Survey 2021.* URL: https://insights.stackoverflow.com/survey/2021. 2021.

[37] Simon Stewart. *Getting Started | Selenium.* URL: https://www.selenium.dev/documentation/webdriver/getting_started/. 2022.

[38] Simon Stewart. *Selenium Webdriver. Proxy.* URL: https://www.selenium.dev/selenium/docs/api/javascript/module/selenium-webdriver/proxy.html. 2022.