BSP03 - Stealthy Browser Fingerprinting

April 15, 2021 - 08:42

Guillaume Tostes
University of Luxembourg
Email: guillaume.tostes.001@student.uni.lu

Project Tutor:
Stefan Schiffner

***Abstract*—This paper presents the third Bachelor Semester Project made by Guillaume TOSTES for his Semester 3 Bachelor in Computer Science at the University of Luxembourg. This project aims to develop a python script able to identify websites using common methods of browser fingerprinting. In the first part, we will take a look at the different techniques available for browser fingerprinting, how they can be implemented and obfuscated. In the second part, we will present the script developed during the semester capable of identifying, in a rough way, if a specific website is using one of those techniques.**

1. Introduction

Nowadays, the main source of income on the internet comes from advertisements. This allows platforms such as YouTube or Facebook to offer their services for free and turn a profit. Therefore such platforms need to engage with their users with relevant advertisements. It has also been shown that providing targeted advertisements improves the user experience and user's approval of advertisements [1]. This has been done for a while using cookies for instance. While browsing different websites each one would place a cookie in the user's computer containing a random number. This number corresponds to a certain user profile in the website's database. Therefore if a customer has items in his cart and a cookie with a random number is put in his computer then the next time the user accesses the website his cart will still be there. This is because the website knows that the user with a cookie containing the number XXXX corresponds to a specific cart. This technology used to work perfectly for advertisers which were able to place perfectly timed ads for users. After browsing an e-commerce website a person is almost 100% sure that he will see an advertisement concerning the website previously browsed. However, there is an issue with cookies, users can manually delete them and many browsers nowadays come with features that automatically block 3rd party cookies.[2]

Since targeted advertisement is worth much more than untargeted advertisement, then advertisement companies developed other ways to target internet users. This new technology is called browser fingerprinting. In short, browser fingerprinting is a method designed to identify unique browsers by creating some sort of "identity" for the browser-based on hundreds of requests made by the website. Since fingerprinting leaves little to no trace on the victim's computer, DNT (Do Not Track) policies are hard to enforce. Checking the DNT box in a browser can even serve to track users as it gives information to websites accessed. [3]
The paper is organized as follows: firstly, the scientific deliverable will be a study on the different fingerprinting methods that are available and how easy it is to hide them. Secondly, the technical deliverable will be a tool developed with python to detect the use of the aforementioned methods.

2. Project description

2.1. Domains

The scientific deliverable will touch upon the field of web development and privacy across the internet. We will be analyzing the possibility to use certain JavaScript methods to perform browser fingerprints. And how code can be obfuscated in order to conduct fingerprinting without being detected.

The technical deliverable will revolve around web scraping, a term that describes the act of extracting data from websites. Extracted data can range from images, sound files to script files. The aforementioned web scraping will be done using the "requests" and "BeautifulSoup4" libraries for Python.

2.2. Targeted Deliverables

The expected scientific deliverable will be a study where firstly, various fingerprinting techniques will be presented according to certain criteria and their advantages will be explained. Secondly, the study will focus on the ability to hide the behavior of this technique from the user.

The expected technical deliverable will be a tool capable of scanning a list of websites passed as parameters using a .txt configuration file and look inside of the Javascript of each if it can find the methods/APIs located in another .txt configuration file.

3. Pre-requisites

The only pre-requisite to comprehend the extent of the scientific deliverable is the knowledge of some web development terminology and be familiar with website behaviors. Understanding what are APIs and how they work previous to start reading this paper is recommended, however, an explanation will be provided.

In order to understand the extent of the technical deliverable, it is highly recommended to be familiar with the python programming language and machine learning. Moreover, to run the python tool it is necessary to have at least Python 3.8.0 installed. Furthermore, the following libraries are required:

```python
import os, shutil, time, sys
import requests
from urllib.parse import urljoin
from bs4 import BeautifulSoup
import re, csv
import get_websites
```

It is necessary to install the "requests" and "bs4" libraries using commands "pip3 install requests" and "pip3 install bs4" respectively.

4. Browser fingerprinting implementation and the ability to hide it

4.1. Research question

In this section of the report, we will be taking a look at the methods used by developers to implement browser fingerprinting. Although there are many different methods, many of them rely on the same principles. Things get different depending on the scale of the fingerprint desired.

This deliverable will explain what are the common methods used and for each their advantage. We live in a world where many stakes are being disputed by advertisers. Therefore, the one who is able to best target users can sell advertisements for a higher price. This is where the role of browser fingerprinting comes in. Given that old technology used by advertisers is slowly being blocked by third party software or browsers, browser fingerprinting is a new method being used in order to track web users. Fingerprinting relies on "questions" asked by the website to the browser, which can be made using JavaScript. Therefore, if we know what requests are susceptible to translate into a fingerprint then we can scan the website's JavaScript to find such requests. However, developers are aware of that and may try to hide such requests in order to prevent software from blocking or identifying websites that perform fingerprints. Hence, we shall present how browser fingerprinting is implemented but also how fingerprinting behavior can be detected. And then assess the possibility to hide such behavior.

4.2. Related Work

In this subsection, we will present in-depth how browser fingerprinting works and the different methods used in order to implement it. As stated before, browser fingerprinting is a technique used to uniquely identify any web browser and track users online. This is done by performing what is called a fingerprint of the target's web browser. This fingerprint is the result of hundreds if not thousands of requests made by the website to the browser. These requests can be very simple as asking for the display resolution of the browser or as a more complex such as the name of the WebGL renderer. The requests are mainly made through script files that run inside the browser. And for a majority of these requests, no permission from the user is necessary. Browser fingerprinting is considered as a stateless tracking technique as no data is stored in the target's browser, unlike cookies.

A browser fingerprint is then the set of collected information from the target device. It contains information regarding the hardware as well as software configurations from the target. By compiling all the results obtained, the server can create a profile corresponding to the fingerprint of that web browser. This fingerprint can later be transmitted to fingerprint databases that store millions of other fingerprints. Let's say we have a user that goes to website A which performs a browser fingerprint. Now, suppose the user from which the fingerprint was created goes to website B which also performs a fingerprint. Then by cross-referencing the fingerprint from both websites, website B can tell if the user went to website A before. And with that, website B can show relevant advertisements about website A to the user. Which means selling the advertisement spot for a premium.

The term browser fingerprint was coined after the "Panopticlick" experiment by Peter Eckersley[5]. This experiment investigated the uniqueness of web browsers. To that end, the authors developed a fingerprinting algorithm that collected data from hundreds of thousands of users. From the data collected, only one in 286,777 fingerprints is not unique. Furthermore, regarding users that enabled Java or Flash, the uniqueness percentage rose from 83.6% to 94.2%. This study brought forth concerns regarding fingerprinting, as the results clearly indicated that this kind of tracking is very powerful.

However, browser uniqueness can be dated back to the inception of web browsers thanks to the "user agent" header. This header was developed in order to prevent website incompatibilities. It enables devices to communicate what browser they were running on what type of device etc... This lets servers display the correct version of websites, limiting incompatibilities. The "user agent" string is considered the first time information was sent to websites that directly indicated properties of the current device [4]. The User-agent header is still used to this day. Even though some of the elements in the user agent header are considered outdated, it still provides some information that can be used to identify one device from another.

To go further into browser uniqueness and what makes reliable fingerprints, we have to look at how users can customize their browsing experience. Unlike the user-agent agent, a customized browsing experience can provide lots of detail regarding a user. Indeed, the user-agent string will only reveal some technical data that won't reveal much about the user. A fingerprint can be made but no predictions regarding the user can be accurately done. This is where things such as browser extensions can make a real impact on a fingerprint. By trying to have a tailor-made experience while browsing the web, users can end up revealing a lot about themselves involuntarily. Suppose a user has some "gaming" extensions enabled in his browser, then a fingerprint can reveal what type of user he is. In such a case, the fingerprint becomes much more powerful than just tracking the user. It can profile him as well.

However, not all extensions can be detected as no API fetches the list of extensions installed. Workarounds have been tried such as the ones proposed in [7]. The developer fetches the manifest of a certain extension by using its ID from the Chrome Store, if a file is returned then the extension is currently installed. This method did not prove very successful as Google has limited the capability of clients communicating with extensions. However, a second method using web-accessible resources from extensions shows how extensions can be easily detected. A study conducted in 2016 established that out of 12,514 Google Chrome extensions, 43,429 could be identified using this technique [8]. Furthermore, a second study was conducted based on the technique described previously that aimed to detect the uniqueness of 16,393 browsers. The testing results showed that just by analyzing 485 carefully selected extensions, 53.96% of users could be uniquely identified [9] (approximately the same results as for testing all 16,743 extensions detected without selection). Half of the users could be uniquely identified with only one parameter: browser extensions. In this case, we can clearly see the power that fingerprinting has.

Further studies conducted based on the data collected by Panopticlick[5] and AmIUnique[10] have calculated the entropy of the different fingerprinting attributes. The entropy is a measurement to quantify the level of uniqueness of a certain attribute of a fingerprint. The entropy is expressed by the following equation:

$$H(X) = -\sum_{i=0}^{n} P(x_i) \log_2 P(x_i)$$

Attribute	Panopticlick		AmIUnique	
	Entropy	Norm.	Entropy	Norm.
List of Plugins	15.400	0.817	11.060	0.656
Available Fonts	13.900	0.738	8.379	0.497
Canvas	-	-	8.278	0.491
Content language	-	-	5.918	0.351
User-agent	10.000	0.531	9.779	0.580
Screen resolution	4.830	0.256	4.889	0.290
List of HTTP headers	-	-	4.198	0.249
Timezone	3.040	0.161	3.338	0.198

Table 1. COMPUTED ENTROPY AND NORMALIZED ENTROPY OF FINGERPRINTING ATTRIBUTES

Where H(X) is the calculated entropy in bits, X is a discrete random variable, and P(X) is a probability mass function. We get Table 1 thanks to the data collected in [11].

4.3. Approach

We can see in Table 1 the attributes that reveal the most about the uniqueness of a browser. Since they seem to be the most reliable attributes we can assume that fingerprinting will rely on them. We know from section 4.3 that fingerprinting can be performed in the JavaScript of a page. Therefore, fingerprinting depends on requests made by the website to the web browser. In that case, if we find the JavaScript methods that enable developers to fetch that type of data, then we could in a way predict if a website is performing a fingerprint.

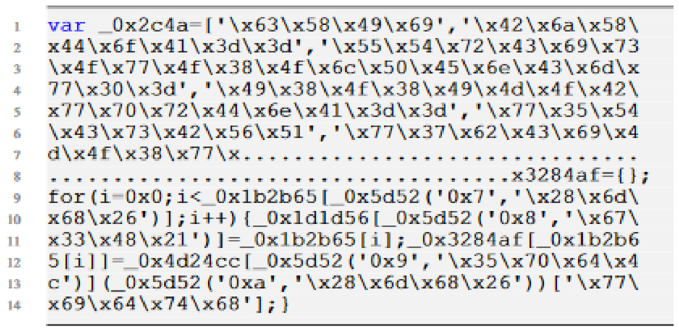
After some time researching, multiple JavaScript methods, as well as one specific term, seem to be linked a lot to fingerprinting. The following list contains common functions used in fingerprinting and is non-exhaustive:

- .plugins
- .javaEnabled
- .getUserMedia
- .availHeight
- .availWidth
- .charCodeAt
- .colorDepth
- .getTimezoneOffset
- .platform
- canvas

From this list, we can clearly spot the methods that are linked to the attributes presented in Table 1. What we can conclude from this list is that these JavaScript functions are not exclusive to fingerprinting, meaning that they are used outside of that context. For example, the .getUserMedia function is used to detect if the user has a microphone or webcam. This can be used on websites that require such hardware. So how come it is also used for fingerprinting?

Some of these methods seem harmless and pretty straightforward. One would not relate them to web-tracking at first glance. This is where the appeal of fingerprinting appears. Since the methods called are pretty harmless, they don't pose any direct privacy threat. Once they are all combined and form a decent data-set then they carry a tracking potential. Furthermore, the fact that these are mundane JavaScript methods then it is necessary to identify manually the purpose they serve in a web-page before being able to confirm if there is malicious intent. As they are, for the most part, used to provide an optimal user experience.

This is where we start to ponder on the potential of hiding fingerprinting behavior. Indeed, as stated in the previous paragraph, it is necessary to manually inspect the source code to determine the intent of a JavaScript method. This seems like a lot of work in order to identify fingerprinting websites. Even more so when we think of code obfuscation or plain bad coding. An enemy of a manual approach is code readability. Developers could obfuscate code in purpose to make it hard to identify fingerprinting behavior. Figure 3.a) of [18] illustrates this example perfectly:



```
var _0x2c4a=['\x63\x58\x49\x69','\x42\x6a\x58\
x44\x6f\x41\x3d\x3d','\x55\x54\x72\x43\x69\x73
\x4f\x77\x4f\x38\x4f\x6c\x50\x45\x6e\x43\x6d\x
77\x30\x3d','\x49\x38\x4f\x38\x49\x4d\x4f\x42\
x77\x70\x72\x44\x6e\x41\x3d\x3d','\x77\x35\x54
\x43\x73\x42\x56\x51','\x77\x37\x62\x43\x69\x4
d\x4f\x38\x77\x.................................
..................................x3284af={};
for(i=0x0;i<_0x1b2b65[_0x5d52('0x7','\x28\x6d\
x68\x26')];i++){_0x1d1d56[_0x5d52('0x8','\x67\
x33\x48\x21')]=_0x1b2b65[i];_0x3284af[_0x1b2b6
5[i]]=_0x4d24cc[_0x5d52('0x9','\x35\x70\x64\x4
c')](_0x5d52('0xa','\x28\x6d\x68\x26'))['\x77\
x69\x64\x74\x68'];}
```

(a) Obfuscated canvas font fingerprinting script from Script 2.

Figure 1. Image taken from Figure 3.a) of [18] all credit goes to the authors of the cited paper.

However, what we could do instead is predict fingerprinting behavior based on previously known data. As the entropy of the different attributes has been calculated in 1 it showed that they play a major role in fingerprinting Then we could predict fingerprinting by looking for a combination of the methods found. If a web-page possesses multiple functions related to attributes that carry high entropy, then we could expect that some kind of fingerprinting is happening.

To that end, we will develop a tool that will be able to detect the presence of certain JavaScript methods in multiple websites. The tool will be presented in-depth in Section 5. With it, we could be able to suspect, or not, fingerprinting behavior in certain web-pages. Web-pages that contain many of the aforementioned methods would be identifiable from the others. The tool in itself will only be a data collection tool. It won't predict anything by itself. It should output the number of times the JavaScript methods appear in the source code. From the data collected, we could then consider developing a Machine Learning algorithm that would predict fingerprinting based on an existing list of fingerprinting website, used as the baseline.

4.4. Results

In this subsection, we will assess the results obtained from the Python Tool developed during the semester. It is recommended but not required that Section 5 has been read prior to continuing Section 4.4.

With our, Python Tool implemented we are now able to detect certain JavaScript functions in the source code of specific websites. The list of methods we look for can be altered manually anytime. Allowing us to test different parameters. The websites in which we decided to run the tool are Alexa's top 1M list. This list contains the top websites in Alexa's traffic rank. This list is quite useful as we were able to run our tool on websites that are the most requested on the internet and get a behind the scenes look at the most common websites on the internet. For the sake of time, we only chose to run the tool on the top 100 websites of Alexa's 1M list. Execution time is rather long and counted in minutes since all websites are checked sequentially and not in parallel.

Figure 12 shows an extract of the resulting ".csv", sorted from most ocurrences to less.

By looking at Figure 12 we cannot seem to recognize any clear pattern apart from the fact that some of the methods have more occurrences than others. As of right now unfortunately the results are not exploitable to the naked eye as some of the websites do not allow the tool to download the source code (e.g. "google.com") and a raw output like the one we obtain doesn't have the same power as statistics. However, this type of output would be great to couple with a machine learning algorithm. We could consider using the data collected to create a k-nearest neighbor algorithm based on a list of websites that conduct fingerprinting in order to establish some kind of relationship between our output and the presence of fingerprinting. Indeed, our tool doesn't serve as a standalone fingerprint detector but serves rather as a data collector for machine learning.

4.5. Conclusion

As we previously stated, our approach doesn't allow us to to identify fingerprinting as easily as we wanted to. Indeed, the tool is only capable of detecting certain methods that are related to fingerprinting. An issue in this approach is that fingerprinting techniques are always evolving. Therefore, the tool could quickly become outdated.

Furthermore, code obfuscation is an enemy of our Python Tool since the tool looks for specific keywords in the source files. If for some reason the keywords are obfuscated in the code in some way, then the tool won't be able to detect it.

However, by crossing our data with the entropy of fingerprinting attributes we could compute how much entropy the source code of a page carries and estimate fingerprinting potential.

5. Python tool for Browser Fingerprint detection

5.1. Requirements

In this section of the report, we will present the tool that was developed during the semester. This tool is supposed to identify the presence of fingerprinting techniques in a website's source code. It has been developed using python and the BeautifulSoup library.

By going through the source code of a website the tool should be able to extract the source code. From there it should be able to analyze the JavaScript code in order to identify the presence of certain methods commonly used in browser fingerprinting. The purpose of it is to create a data-set of the methods used in the desired website for later use in Machine Learning algorithms. The tool will only be able to detect fingerprinting written in JavaScript since those are the files that we are interested in for now.

It is required that the tool has the capability to analyze many websites for many JavaScript methods. Therefore, it is necessary that the tool also comes with two configuration files. One for all the websites that the tool will analyze and one for all the JavaScript methods it needs to look for.

Since an unknown number of websites and parameters is expected each time the tool will run, it is important that the tool can run for a long period of time without crashing from errors. Therefore, errors being thrown during execution must be handled in order to continue the process in case of failure in a particular.
Furthermore, there is no way to control in which computer the tool will run. So it will be necessary to make sure that it is runnable many times in different directories to ensure that the tool won't break when used by someone else than the developer.

A requirement that was added later in the development is in regards to the output generated by the tool. It isn't much use if the tool only prints the results in the terminal. Hence, the tool will save all of the collected data in a ".csv" file. The file must be standardized and exploitable at a later date by machine learning algorithms to identify the presence of fingerprinting. Finally, the tool will also come with a read-me file detailing how to setup the tool.

5.2. Design

In order to develop the tool, it was necessary to establish how it would work and plan all the different modules necessary. By going through the requirements previously stated we get a rough idea of what we are supposed to implement. The first aspect of the tool is a web-crawler capability. By passing a website as parameter it should be able to access the website and download a local copy of its source code. The local copy of the website's files should be easily accessible by the tool as well as the user. Therefore we will have to implement some sort of file organizer in order to keep the current directory organized and enable the tool to later retrieve the files properly.

The tool will also need to access both configuration files at startup in order to save all the websites and parameters in their respective arrays for later use. This means it needs to read the configuration files line by line and save each line in the correct variable. The configuration files should respect precise formatting and not contain characters that do not belong in URLs. The rules for writing in both configuration files will be explained in the readme file in order to instruct the user on how to use them. For now we are only interested in JavaScript files. There should be a function capable of deleting all other files from the local copy of a website. The function should appropriately name the directories in which the files are saved.

Now that the tool has all of its basic requirements, we need to think about implementing the heart of the tool. There should be a function capable of parsing through any ".jss" file passed as parameter and check if its contains a certain method, also passed as parameter. This function shall return in some way the number of times the JavaScript method has appeared in the file.

For the main executable of the tool, it should be pretty straightforward what we are supposed to implement. After having created our arrays containing the websites and parameters we should iterate through all of the websites in a "for" loop. In this loop, we will make sure that the website's URL is correctly typed and able to be fetched by the download function. Then it should download the web-page and use the file organizer to properly save the files in the correct directories. Once this is done, it will iterate through all of the parameters in a "for" loop nested in the previous loop. It will use our parsing function to go through all the files of the current website and look for specific methods. To make sure that the tool meets the requirement regarding crash-less executions then we should make sure that it catches exceptions thrown during web-page download and parsing.

To make sure that the tool can run on various machines, we need to make sure that the path it uses for all the file access and management is dynamically declared based on the current path of the python executable. Finally, since we want to write the data in a ".csv" file it is necessary that we properly declare the headers (column names) and properly store in dictionaries all of the data collected for easy write in the file. It should contain the name of each method in the first row and each subsequent row will contain the URL of the website and the occurrence of each method in its code. "CSV" stands for "comma-separated values" and is used to store values in an organized way. Each row contains various fields. These types of files are used in applications that deal will big amounts of data since accessing and editing them is simple and can easily be used together with programming languages such as Python.

5.3. Production

In this subsection, we will explain in depth how the python tool was implemented to follow the design principles we previously stated. The tool is available for download on GitHub in the following link: https://github.com/guillaumetst/BSP03-Web-Scraper-5.0.
The tool is composed of 4 files, 2 of them are python files and the last 2 are text files that serve as configuration files. The 2 python files contain the code to two different programs. The file "get_websites.py" was created in order to download Alexa's top 1 million websites list. This list can be downloaded in a ".zip" file. Therefore it was necessary to create a python script to automate the download, extraction, and formatting of the list to be used in the main program. We download the list in a select location inside the current folder and copy all the URLs in the list into another ".csv" file while removing the indexes and headers.

The second python file contains the main program.

In Figure 2 we can see the "setup(n)" function that will execute at the start of each execution of the tool. This function first retrieves all of the parameters present in the "parameters_config_file.txt" and stores them in a global array while removing return carriage. Next, the function will retrieve the websites it will work with from the file "website_config_fle.txt". Depending on the input parameter passed by the user to the setup function, the function will execute two different actions. If the input parameter is equal to 0, the function will retrieve the websites from the user-created file in the same directory. Else, if the input parameter is equal to a number n different from 0 then the function will retrieve the n top websites from the "Alexa Top 1M" list. For the second case, we had to implement a "get_top_websites()" function in another file. This is because the list is provided in a zip file. Therefore, it was necessary to automate the process of downloading the file, extracting it, and formatting the data correctly for it to be compatible with the previously designed tool. The functions designed for this purpose are available in Figure 3 in the Appendix.

Before doing any manipulations we have to make sure that we start with a clean slate with each execution of the tool. To that end, a cleanup function was developed in order to delete any files from a previous execution if there are any. Figure 4 in the Appendix contains the cleanup function and its different outcomes.

To talk about the next function defined we have to explain the concepts behind the "BeautifulSoup" and "requests" libraries. The "requests" library was designed in order to perform HTTP requests using python. It enables the developer to create a request with a single line of code, facilitating such actions. We will be using this library to create HTTP requests to the desired websites.
The second important library used is "BeautifulSoup". This library is used for web-scraping purposes and enables us to extract information from web-pages. We will be using this to extract all the files we need from the desired web-pages.

In Figure 5 we can get a look at the function that downloads web-pages for us. The first thing we do in the "savePage()" function is to declare a local "soupfindnSave()" function. The local function will look for the specified type of file in the "soup" that we give to it. In our case, we are looking for script files. Basically, the "soupfindnSave()" function will save all of the script files found of the given "soup" into a folder given by the "pagefolder" parameter. Before calling this function, first, we need to create an HTTP request that will enable us to get to the website. To that end we create a session variable using the "requests" library. Then we use the ".get(url)" method in order to make the HTTP request to the server and get the response from the website. The next step is to create a "soup" of the response we got using the "BeautifulSoup" library. The soup created will then be passed as a parameter to the previously declared "soupfindnSave()" function. Once the function is done, we obtain all of the script files of the webpage we desire.

After having downloaded the data we needed, we have to perform one last operation on the files. Figure 6 shows the "js_filter()" function that was developed to make sure that all of the files are indeed javascript by looking for the ".js" extension. And then correctly moving them from a temporary folder to the WebPageDownloads folder which will hold all of the files in their respective folder, named after the website it originates from.

Once all of the setting up is done we are now able to analyze the contents of the downloaded files. To that end the last function called "js_parser()" was developed, see the following image for the first part of the function.

We will split the function into two since the first part is where it searches for a string in a file. The second part is just for making a user-friendly output file (which is now outdated given the latest implementations). This function needs two inputs to work. It needs the path to a file and a string to look for. It starts by creating a "summary.txt" file for the user-friendly output. Then it initializes a variable that will be used to count occurrences of the string and an array that will contain text outputs regarding the presence of the string in a file. The contents of the array will be written in a text file later on. Next, the function goes through all of the files located in the given path and creates for each a file path and file name. Then it opens each file and looks for the string that was passed as an input parameter while making sure that it doesn't open the "summary.txt" file created at the start. If the string is present in the file then we add 1 to the "appearance_counter" and add the name of the file to the "appeared_in" array.

The rest of the function is available in the appendix in Figure 8. In the second part the function will write the current outcome to a summary file. Enabling the user to see which files contain which strings manually. Finally, the function returns the number of times the string appeared in the file.

Now that all preliminary work is done we can explain the main execution of our tool. Figures 9, 10 and 11 show the main program that will run.

Firstly the "setup(n)" function is called and we obtain our arrays of websites and parameters to work with. Then the "folder_cleanup()" function is called to remove the previous files downloaded in an older execution. Next, we create our output file, which will be in a ".csv" format and we create its header by using the name of the JavaScript methods contained in "parameters_array". We end the first part by declaring an array that will contain all of the rows that we will write to the ".csv" file.

Next, we iterate through all of the websites using a "for" loop. From the URL of each website, we extract the domain name. We also create a dictionary that will hold the URL as well as the number of occurrences of each parameter we are looking for. Now, the tool will have to download the files from the current website and analyze them. To that end we create a "try:" statement that encapsulates the "savePage()" function call. This is to make sure we catch any errors thrown in case the tool isn't able to access a website. We catch any errors with an "except:" statement in which we skip the current website. Going back to the "try" statement, the program will save a web-page as well as look for the presence of certain JavaScript methods. The results of the "js_parser()" function calls are saved in a dictionary containing the name of the JavaScript method as well as the number of appearances. The dictionary is then appended to the array that will contain all the rows of our output ".csv" file.

Finally, we end the program by opening our output file and writing the header followed by all the rows stored in the "row_array" variable (refer to Figure 11).

5.4. Assessment

During the semester we were able to match all of the functional and non-functional requirements specified in Section 5.1. Indeed, the tool matches our expectations. However, even though the requirements have been met it doesn't mean the developed tool is optimal. Right now we are able to identify the presence of specific JavaScript methods in the sources. The program could be improved to identify API requests within HTML and JavaScript files. This could be done by either inspecting HTML source files or looking for specific fingerprinting APIs.

Moreover, the console output could be standardized in a way to correspond to UNIX system standards such that the standard output of the tool can be passed as standard input of another program using the pipe character ("—"). However, since the tool was made for Windows and I had no prior knowledge of such implementation this was not set as a requirement.

Regarding the output file, it was decided to output a ".csv" to use it for Machine Learning. Indeed, a next step in the project would be to develop a machine learning algorithm that is able to detect if a certain website performs fingerprinting. We could use a k-nearest neighbor algorithm in order to predict fingerprinting. This would be based on the collected data from the python tool developed this semester. For that, we still need a reliable source that can confirm if a website actually performs a browser fingerprint.

Acknowledgment

The author would like to thank the BiCS management and Stefan SCHIFFNER for the incredible support provided during the whole duration of the semester. The author would also like to thank Abdelwahab BOUALOUACHE for participating in the grading of this project.

6. Conclusion

In this paper, we have presented what browser fingerprinting is, its purpose, and how it can be used to track online users. We have also explained how it can be implemented and hidden. The scientific question posed at the start of the paper regarded the possibility to hide fingerprinting from detectors. To that end, we came up with an approach capable of identifying fingerprinting behavior on a small scale. In our approach, we aimed to identify potential fingerprinting behavior but not to classify websites. The scientific part of this project is directly linked to the technical deliverable which consisted in developing a Python Tool capable of identifying fingerprinting behavior. For the technical part of the paper, we set requirements which the tool had to fulfill. As a whole, the tool developed fulfilled all of the requirements that were set. The tool will work on various Windows devices and from the tests conducted won't crash during important steps as precautionary measures have been taken.
This semester project was an all-around look into browser fingerprinting and web development. It also served as an initiation into Web Scraping and BeautifulSoup. This was

a very interesting project that touched in multiple fields but kept coherence. I can firmly state that I have learned a lot from working on this project and from working with my project tutor, Stefan SCHIFFNER, who taught me a lot about the subject and made me discover a previously unknown side of internet privacy.

References

[1] Jan H. Schumann, Florian von Wangenheim, Nicole Groene *Targeted Online Advertising: Using Reciprocity Appeals to Increase Acceptance among Users of Free Web Services.*
https://journals.sagepub.com/doi/abs/10.1509/jm.11.0316

[2] Michal Wlosik, Michael Sweeney *What's the Difference Between First-Party and Third-Party Cookies?*
https://journals.sagepub.com/doi/abs/10.1509/jm.11.0316

[3] Chris Hoffman *RIP "Do Not Track," the Privacy Standard Everyone Ignored.*
https://www.howtogeek.com/fyi/rip-do-not-track-the-privacy-standard-everyone-ignored/

[4] Nicholas C. Zakas *History of the user-agent string*
https://humanwhocodes.com/blog/2010/01/12/history-of-the-user-agent-string/

[5] Peter Eckersley *How unique is your web browser?*
https://dl.acm.org/doi/10.5555/1881151.1881152

[6] PIERRE LAPERDRIX, NATALIIA BIELOVA, BENOIT BAUDRY, GILDAS AVOINE *Browser Fingerprinting: A survey*
https://humanwhocodes.com/blog/2010/01/12/history-of-the-user-agent-string/

[7] Amir Khashayar Mohammadi *How to Detect Browser Extensions*
https://blog.authentic8.com/how-to-detect-browser-extensions/

[8] Alexander Sjösten, Steven Van Acker, Andrei Sabelfeld *Discovering Browser Extensions viaWeb Accessible Resources*
https://www.cse.chalmers.se/research/group/security/publications/2017/extensions/codaspy-17.pdf

[9] Gábor György Gulyás, Dolière Francis Somé, Nataliia Bielova, Claude Castelluccia *To Extend or not to Extend: on the Uniqueness of BrowserExtensions and Web Logins*
https://www-sop.inria.fr/members/Nataliia.Bielova/papers/Guly-etal-18-WPES.pdf

[10] Pierre Laperdrix *AmIUnique*
https://www.amiunique.org/

[11] Alejandro Gómez-Boix, Pierre Laperdrix, Benoit Baudry *Hiding in the Crowd: an Analysis of the Efectiveness of BrowserFingerprinting at Large Scale*
https://331.cybersec.fun/EffectivenessOfFingerprinting.pdf

[12] Julia Kho *How to Web Scrape with Python in 4 Minutes*
https://towardsdatascience.com/how-to-web-scrape-with-python-in-4-minutes-bc49186a8460

[13] *Beautiful Soup Documentation*
https://www.crummy.com/software/BeautifulSoup/bs4/doc/

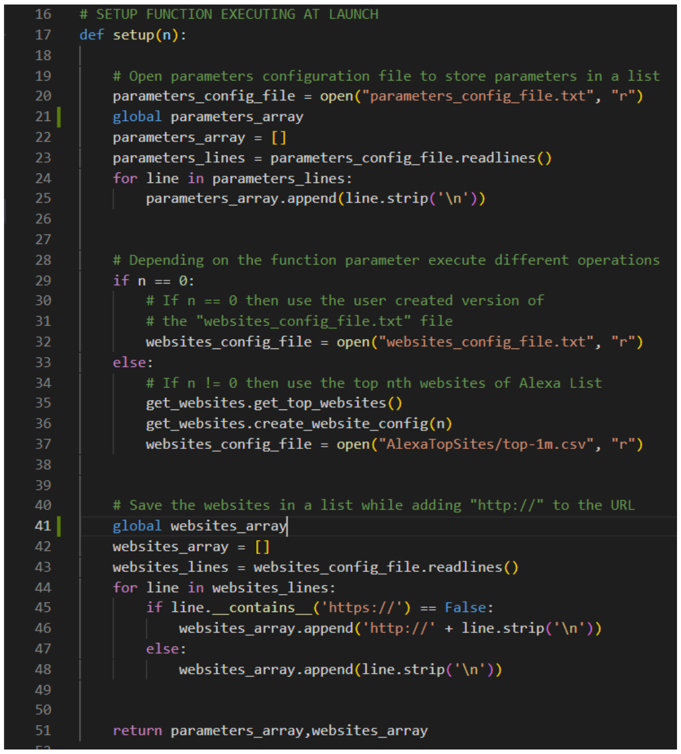
[14] *FingerprintJS*
https://fingerprintjs.com/

[15] Scott Stensland *Access microphone from a browser - Javascript*
https://stackoverflow.com/questions/27846392/access-microphone-from-a-browser-javascript

[16] Umar Iqbal, Steven Englehardt, Zubair Shafiq *FP-Inspector*
https://uiowa-irl.github.io/FP-Inspector/

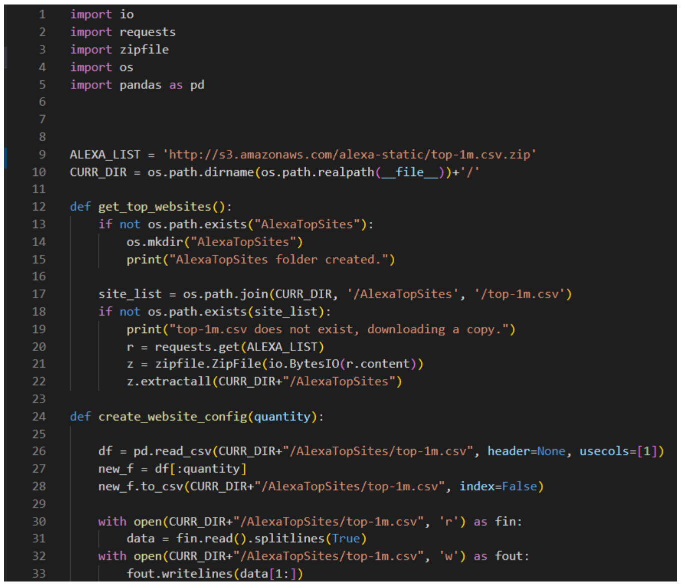
[17] Peter Hraška *We've analysed 500,000 browser fingerprints. Here is what we found.*
https://medium.com/slido-dev-blog/we-collected-500-000-browser-fingerprints-here-is-what-we-found-82c319464dc9

[18] Umar Iqbal, Steven Englehardt, Zubair Shafiq *Fingerprinting the Fingerprinters:Learning to Detect Browser Fingerprinting Behaviors*
https://umariqbal.com/papers/fpinspector-sp2021.pdf



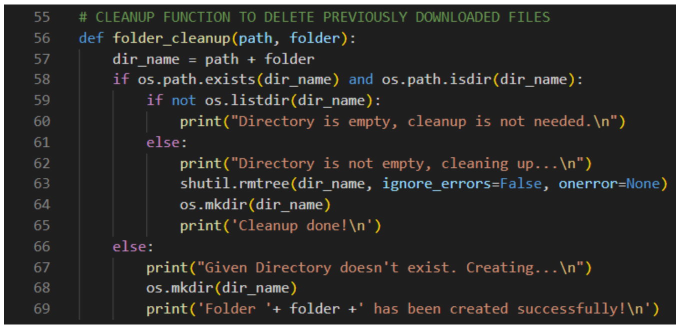
```python
# SETUP FUNCTION EXECUTING AT LAUNCH
def setup(n):

    # Open parameters configuration file to store parameters in a list
    parameters_config_file = open("parameters_config_file.txt", "r")
    global parameters_array
    parameters_array = []
    parameters_lines = parameters_config_file.readlines()
    for line in parameters_lines:
        parameters_array.append(line.strip('\n'))


    # Depending on the function parameter execute different operations
    if n == 0:
        # If n == 0 then use the user created version of
        # the "websites_config_file.txt" file
        websites_config_file = open("websites_config_file.txt", "r")
    else:
        # If n != 0 then use the top nth websites of Alexa List
        get_websites.get_top_websites()
        get_websites.create_website_config(n)
        websites_config_file = open("AlexaTopSites/top-1m.csv", "r")


    # Save the websites in a list while adding "http://" to the URL
    global websites_array
    websites_array = []
    websites_lines = websites_config_file.readlines()
    for line in websites_lines:
        if line.__contains__('https://') == False:
            websites_array.append('http://' + line.strip('\n'))
        else:
            websites_array.append(line.strip('\n'))


    return parameters_array,websites_array
```

Figure 2. Setup function



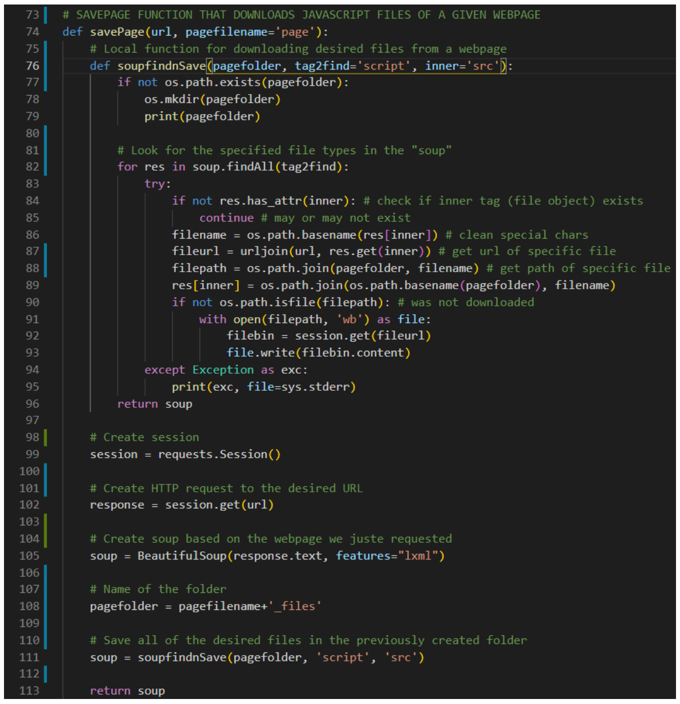
```python
import io
import requests
import zipfile
import os
import pandas as pd



ALEXA_LIST = 'http://s3.amazonaws.com/alexa-static/top-1m.csv.zip'
CURR_DIR = os.path.dirname(os.path.realpath(__file__))+'/'

def get_top_websites():
    if not os.path.exists("AlexaTopSites"):
        os.mkdir("AlexaTopSites")
        print("AlexaTopSites folder created.")

    site_list = os.path.join(CURR_DIR, '/AlexaTopSites', '/top-1m.csv')
    if not os.path.exists(site_list):
        print("top-1m.csv does not exist, downloading a copy.")
        r = requests.get(ALEXA_LIST)
        z = zipfile.ZipFile(io.BytesIO(r.content))
        z.extractall(CURR_DIR+"/AlexaTopSites")

def create_website_config(quantity):

    df = pd.read_csv(CURR_DIR+"/AlexaTopSites/top-1m.csv", header=None, usecols=[1])
    new_f = df[:quantity]
    new_f.to_csv(CURR_DIR+"/AlexaTopSites/top-1m.csv", index=False)

    with open(CURR_DIR+"/AlexaTopSites/top-1m.csv", 'r') as fin:
        data = fin.read().splitlines(True)
    with open(CURR_DIR+"/AlexaTopSites/top-1m.csv", 'w') as fout:
        fout.writelines(data[1:])
```

Figure 3. get_websites() function



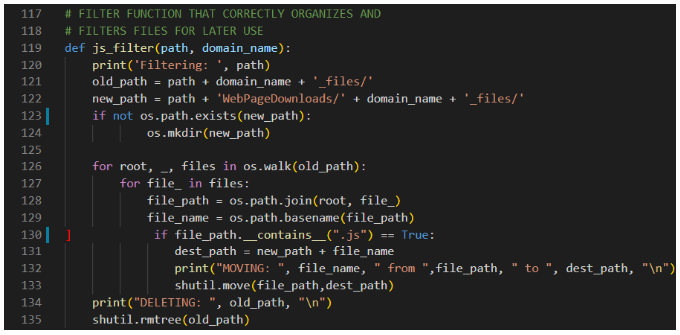
```python
# CLEANUP FUNCTION TO DELETE PREVIOUSLY DOWNLOADED FILES
def folder_cleanup(path, folder):
    dir_name = path + folder
    if os.path.exists(dir_name) and os.path.isdir(dir_name):
        if not os.listdir(dir_name):
            print("Directory is empty, cleanup is not needed.\n")
        else:
            print("Directory is not empty, cleaning up...\n")
            shutil.rmtree(dir_name, ignore_errors=False, onerror=None)
            os.mkdir(dir_name)
            print('Cleanup done!\n')
    else:
        print("Given Directory doesn't exist. Creating...\n")
        os.mkdir(dir_name)
        print('Folder '+ folder +' has been created successfully!\n')
```

Figure 4. folder_cleanup() function



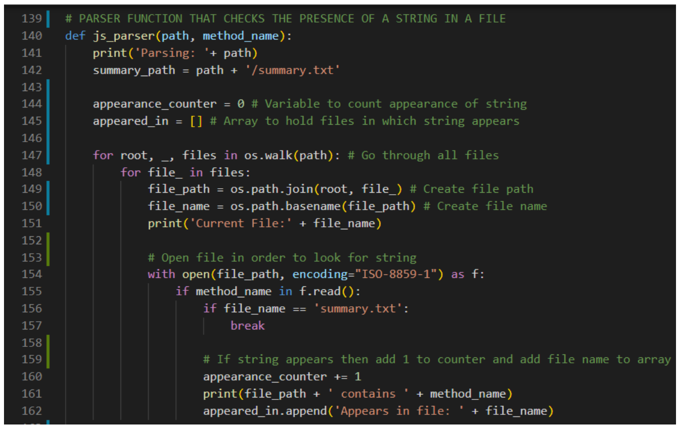
```python
# SAVEPAGE FUNCTION THAT DOWNLOADS JAVASCRIPT FILES OF A GIVEN WEBPAGE
def savePage(url, pagefilename='page'):
    # Local function for downloading desired files from a webpage
    def soupfindnSave(pagefolder, tag2find='script', inner='src'):
        if not os.path.exists(pagefolder):
            os.mkdir(pagefolder)
            print(pagefolder)

        # Look for the specified file types in the "soup"
        for res in soup.findAll(tag2find):
            try:
                if not res.has_attr(inner): # check if inner tag (file object) exists
                    continue # may or may not exist
                filename = os.path.basename(res[inner]) # clean special chars
                fileurl = urljoin(url, res.get(inner)) # get url of specific file
                filepath = os.path.join(pagefolder, filename) # get path of specific file
                res[inner] = os.path.join(os.path.basename(pagefolder), filename)
                if not os.path.isfile(filepath): # was not downloaded
                    with open(filepath, 'wb') as file:
                        filebin = session.get(fileurl)
                        file.write(filebin.content)
            except Exception as exc:
                print(exc, file=sys.stderr)
        return soup

    # Create session
    session = requests.Session()

    # Create HTTP request to the desired URL
    response = session.get(url)

    # Create soup based on the webpage we juste requested
    soup = BeautifulSoup(response.text, features="lxml")

    # Name of the folder
    pagefolder = pagefilename+'_files'

    # Save all of the desired files in the previously created folder
    soup = soupfindnSave(pagefolder, 'script', 'src')

    return soup
```

Figure 5. savePage() function



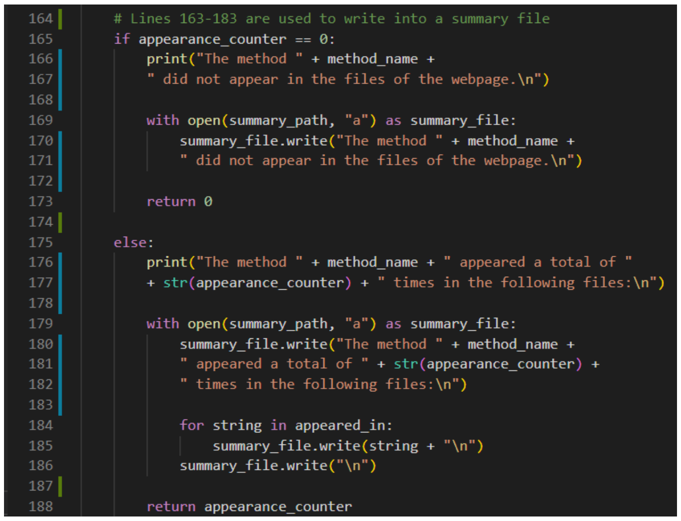
```python
# FILTER FUNCTION THAT CORRECTLY ORGANIZES AND
# FILTERS FILES FOR LATER USE
def js_filter(path, domain_name):
    print('Filtering: ', path)
    old_path = path + domain_name + '_files/'
    new_path = path + 'WebPageDownloads/' + domain_name + '_files/'
    if not os.path.exists(new_path):
            os.mkdir(new_path)

    for root, _, files in os.walk(old_path):
        for file_ in files:
            file_path = os.path.join(root, file_)
            file_name = os.path.basename(file_path)
             if file_path.__contains__(".js") == True:
                dest_path = new_path + file_name
                print("MOVING: ", file_name, " from ",file_path, " to ", dest_path, "\n")
                shutil.move(file_path,dest_path)
    print("DELETING: ", old_path, "\n")
    shutil.rmtree(old_path)
```

Figure 6. js_filter() function



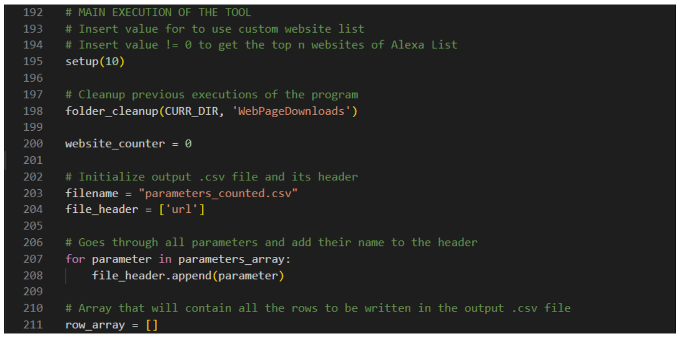
```python
# PARSER FUNCTION THAT CHECKS THE PRESENCE OF A STRING IN A FILE
def js_parser(path, method_name):
    print('Parsing: '+ path)
    summary_path = path + '/summary.txt'

    appearance_counter = 0 # Variable to count appearance of string
    appeared_in = [] # Array to hold files in which string appears

    for root, _, files in os.walk(path): # Go through all files
        for file_ in files:
            file_path = os.path.join(root, file_) # Create file path
            file_name = os.path.basename(file_path) # Create file name
            print('Current File:' + file_name)

            # Open file in order to look for string
            with open(file_path, encoding="ISO-8859-1") as f:
                if method_name in f.read():
                    if file_name == 'summary.txt':
                        break

                    # If string appears then add 1 to counter and add file name to array
                    appearance_counter += 1
                    print(file_path + ' contains ' + method_name)
                    appeared_in.append('Appears in file: ' + file_name)
```

Figure 7. Part 1 of js_parser() function



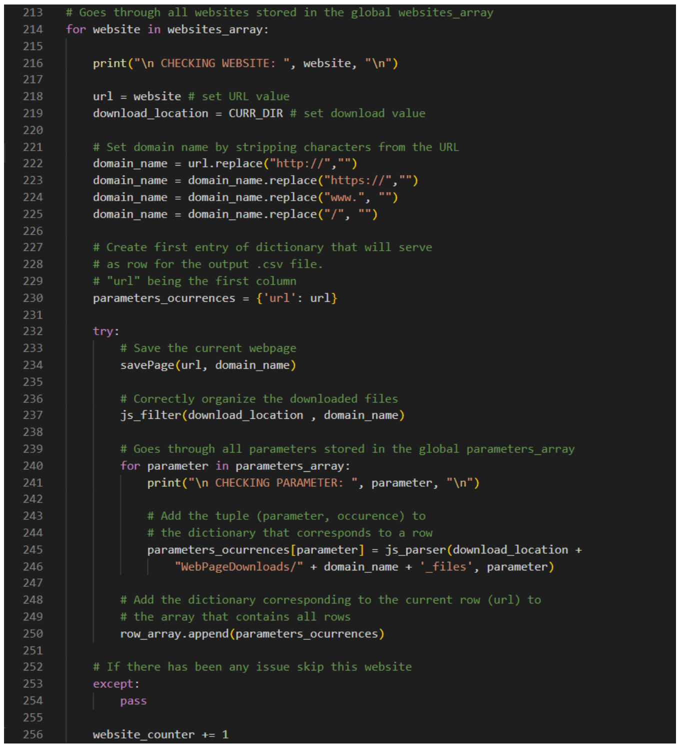
```python
    # Lines 163-183 are used to write into a summary file
    if appearance_counter == 0:
        print("The method " + method_name +
        " did not appear in the files of the webpage.\n")

        with open(summary_path, "a") as summary_file:
            summary_file.write("The method " + method_name +
            " did not appear in the files of the webpage.\n")

        return 0

    else:
        print("The method " + method_name + " appeared a total of "
        + str(appearance_counter) + " times in the following files:\n")

        with open(summary_path, "a") as summary_file:
            summary_file.write("The method " + method_name +
            " appeared a total of " + str(appearance_counter) +
            " times in the following files:\n")

            for string in appeared_in:
                summary_file.write(string + "\n")
            summary_file.write("\n")

        return appearance_counter
```

Figure 8. Part 2 of js_parser() function



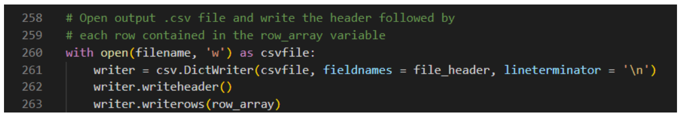
```python
# MAIN EXECUTION OF THE TOOL
# Insert value for to use custom website list
# Insert value != 0 to get the top n websites of Alexa List
setup(10)

# Cleanup previous executions of the program
folder_cleanup(CURR_DIR, 'WebPageDownloads')

website_counter = 0

# Initialize output .csv file and its header
filename = "parameters_counted.csv"
file_header = ['url']

# Goes through all parameters and add their name to the header
for parameter in parameters_array:
    file_header.append(parameter)

# Array that will contain all the rows to be written in the output .csv file
row_array = []
```

Figure 9. Part 1 of main execution of the tool



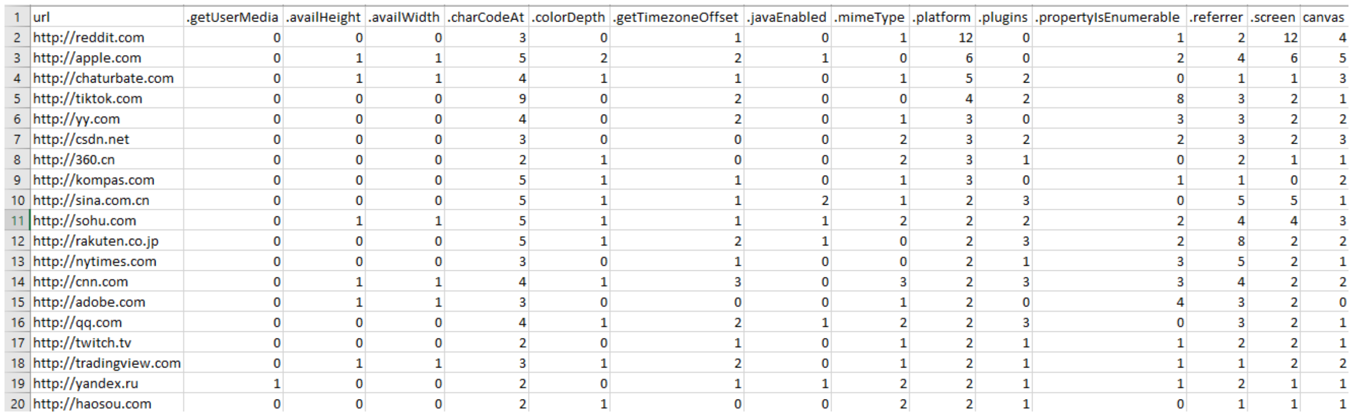
```python
# Goes through all websites stored in the global websites_array
for website in websites_array:

    print("\n CHECKING WEBSITE: ", website, "\n")

    url = website # set URL value
    download_location = CURR_DIR # set download value

    # Set domain name by stripping characters from the URL
    domain_name = url.replace("http://","")
    domain_name = domain_name.replace("https://","")
    domain_name = domain_name.replace("www.", "")
    domain_name = domain_name.replace("/", "")

    # Create first entry of dictionary that will serve
    # as row for the output .csv file.
    # "url" being the first column
    parameters_ocurrences = {'url': url}

    try:
        # Save the current webpage
        savePage(url, domain_name)

        # Correctly organize the downloaded files
        js_filter(download_location , domain_name)

        # Goes through all parameters stored in the global parameters_array
        for parameter in parameters_array:
            print("\n CHECKING PARAMETER: ", parameter, "\n")

            # Add the tuple (parameter, occurence) to
            # the dictionary that corresponds to a row
            parameters_ocurrences[parameter] = js_parser(download_location +
                "WebPageDownloads/" + domain_name + '_files', parameter)

        # Add the dictionary corresponding to the current row (url) to
        # the array that contains all rows
        row_array.append(parameters_ocurrences)

    # If there has been any issue skip this website
    except:
        pass

    website_counter += 1
```

Figure 10. Part 2 of main execution of the tool



```python
# Open output .csv file and write the header followed by
# each row contained in the row_array variable
with open(filename, 'w') as csvfile:
    writer = csv.DictWriter(csvfile, fieldnames = file_header, lineterminator = '\n')
    writer.writeheader()
    writer.writerows(row_array)
```

Figure 11. Part 3 of main execution of the tool



1	url	.getUserMedia	.availHeight	.availWidth	.charCodeAt	.colorDepth	.getTimezoneOffset	.javaEnabled	.mimeType	.platform	.plugins	.propertyIsEnumerable	.referrer	.screen	canvas
2	http://reddit.com	0	0	0	3	0	1	0	1	12	0	1	2	12	4
3	http://apple.com	0	1	1	5	2	2	1	0	6	0	2	4	6	5
4	http://chaturbate.com	0	1	1	4	1	1	0	1	5	2	0	1	1	3
5	http://tiktok.com	0	0	0	9	0	2	0	0	4	2	8	3	2	1
6	http://yy.com	0	0	0	4	0	2	0	1	3	0	3	3	2	2
7	http://csdn.net	0	0	0	3	0	0	0	2	3	2	2	3	2	3
8	http://360.cn	0	0	0	2	1	0	0	2	3	1	0	2	1	1
9	http://kompas.com	0	0	0	5	1	1	0	1	3	0	1	1	0	2
10	http://sina.com.cn	0	0	0	5	1	1	2	1	2	3	0	5	5	1
11	http://sohu.com	0	1	1	5	1	1	1	2	2	2	2	4	4	3
12	http://rakuten.co.jp	0	0	0	5	1	2	1	0	2	3	2	8	2	2
13	http://nytimes.com	0	0	0	3	0	1	0	0	2	1	3	5	2	1
14	http://cnn.com	0	1	1	4	1	3	0	3	2	3	3	4	2	2
15	http://adobe.com	0	1	1	3	0	0	0	1	2	0	4	3	2	0
16	http://qq.com	0	0	0	4	1	2	1	2	2	3	0	3	2	1
17	http://twitch.tv	0	0	0	2	0	1	0	1	2	1	1	2	2	1
18	http://tradingview.com	0	1	1	3	1	2	0	1	2	1	1	1	2	2
19	http://yandex.ru	1	0	0	2	0	1	1	2	2	1	1	2	1	1
20	http://haosou.com	0	0	0	2	1	0	0	2	2	1	0	1	1	1

Figure 12. Output ".csv" sorted.