Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Networked Systems and Services
Laboratory of Cryptography and System Security

Kristof Tamas

Detection of Malicious Web Pages With Static Analysis

SUPERVISOR

Dr. Levente Buttyan

BUDAPEST, 2017

Table of contents

Összefoglaló

A Világháló elterjedtsége miatt manapság a legtöbb ember számára napi rutinná vált az internetezés. Ezt a tevékenységet nem csak PC-n végezhetjük, hanem laptopon, telefonon, tableten, vagy akár egy autó számítógépén keresztül is. A nagyméretű felhasználói bázisnak és a sok, különböző típusú platformnak köszönhetően a web a rosszindulatú programok (malware-ek) egyik elsődleges terjesztőjévé vált. A gyanútlan felhasználó egy nem frissített böngészővel akár néhány weblap meglátogatása után is megfertőződhet anélkül, hogy azt észrevenné.

A kártékony weboldalak detektálására a legelterjedtebb módszer a dinamikus elemzés. Ezek a módszerek a weboldalt egy izolált, gyakran virtuális környezetben töltik be, és vizsgálják, hogy történt-e fájlletöltés, vagy egyéb gyanús változás a számítógépen. A probléma az, hogy az ilyen dinamikus elemzés időigényes. A virtuális környezet felállítása és a weblapon lévő szkriptek lefuttatása akár néhány percet is igénybe vehet. Ez a transzparens használathoz túl lassú, illetve ezzel a módszerrel csak pár ezer oldalt lehet megvizsgálni naponta, ami kevés.

A 2010-es év környékén több cikk jelent meg olyan módszerekről, amelyek egy HTML, vagy JavaScript kódról futtatás nélkül, csupán a statikus jellemzők alapján eldöntik, hogy kártékony vagy sem, így akár több millió weboldalt lehet elemezni naponta. Az elmúlt pár évben azonban jelentősebb kutatási eredmény nem jelent meg a témában és az implementációk sem érhetőek el. A munkám célja ezért az, hogy a korábbi eredményekből kiindulva egy olyan statikus weboldal szűrőt hozzak létre, amely előszűrőként alkalmazható egy alaposabb, de lassabb dinamikus elemző szoftver számára. A megoldásom böngészőbe is ágyazható, ebben az esetben figyelmezteti a felhasználót az esetleges veszélyekre.

Munkám gépi tanulási módszerekre épül, ezért először ártalmatlan és kártékony kódokat gyűjtöttem össze, amelyek a tanító adathalmazt adják. Ezt követően a mintákon kiszámítottam az egyes statikus jellemzők értékeit és egy osztályozót tanítottam be velük. Végül elkészítettem a szűrőt, amely weboldalakat látogat meg, letölti az ott található kódot, kiszámítja a statikus paramétereket, majd elvégzi az osztályozást. Az eredmény annak a valószínűsége, hogy az oldal kártékony.

Abstract

As the World Wide Web is becoming more widespread, nowadays, surfing on the web has become a daily routine for most people. Besides PC, we can use laptops, phones, tablets or even a car's computer to access the internet. Because of the huge user base and the diversity of platforms, the web has become the primary technology for distributing malicious software (malware). An unsuspicious user with an unpatched browser can be infected by a malware without noticing it by merely visiting a few web pages.

The most common way to detect malicious pages is dynamic analysis. These methods load the web pages in isolated environments and search for downloaded files and other suspicious activities. The problem with dynamic analysis is that it's very time-consuming. Setting up the virtual environment and running the scripts on the page can take minutes. This is too slow for transparent usage and we can only analyse a few thousand pages per day with these techniques.

Around 2010 a number of articles appeared about static analysis. Static analysis tries to classify the HTML or JavaScript sources only by their static features, without running them, reaching millions of analysed pages per day. But there are no follow-up research results within the last few years and the implementations of the previous projects are unavailable. Therefore, the goal of my work is to create a static web page filter using the research findings. This filter can be a prefilter for a more sophisticated, but slower dynamic analyser tool, or it can be integrated in a browser to warn the user about possible threats.

My work is based on machine learning techniques, so first, I collected benign and malicious samples, which provide the training set. Then, I calculated the static features of the samples and trained a classifier with them. Finally, I created the filter which crawls web pages, computes the features of the pages, and performs the classification. The result is the probability that the site is malicious.

1 Introduction

In 2017 around half of the Earth's population has Internet connection and can connect to the Internet daily via any device type, which is almost 4 billion users[1,2]. This number is still growing exponentially. One of the most popular applications of the Internet is the World Wide Web, invented by Tim Berners-Lee in 1989 and released to the general public in 1991. To access any content on the Web, we can use not only personal computers and laptops, but also phones, gaming consoles, TVs or even smart cars. Because of the huge user base and the variety of platforms used for web browsing, the Web has become the main application for spreading malicious software (malware). Furthermore, there is no need for the attacker to physically access the victim's device. Without any technical knowledge or skills, the user can be infected without noticing it. A malware can steal personal information and user credentials, or it can lock the user out of the system letting them back only if a specified ransom is paid (ransomware). The lack of user knowledge, the wide accessibility of the internet and the huge impact of an infection result in the overgrowing need for better protection.

1.1 Distributing Malware

There are five possible ways of distributing a malware [2][3], via:

- *Email attachments*: Sending a malicious URL or a malware as an email attachment is easy to perform, but the victim may become suspicious and refuse to download the attachment. Even spam detectors will possibly detect these mails, and will warn the users.
- *Infected storage devices*: The attacker can install a malware from a USB drive, or CD if they access the device physically, which usually requires the attacker to be trusted by the victim.
- *File sharing protocols (FTP, P2P)*: It is a popular technique, but most internet users don't use these protocols or programs.

---

[1] http://www.internetlivestats.com/internet-users/

[2] http://www.internetworldstats.com/stats.htm

- *Malvertising, or malicious advertising*: It downloads malware to a victim's device when the victim loads a web page that displays the malicious advertisement.
- *Malicious web pages*: The JavaScript code on a page can exploit a browser or a plugin vulnerability (e.g., bad memory allocation, buffer overflow) to execute a shellcode defined by the attacker.

During my work, I concentrated on malicious web pages, because one of the most common methods to distribute malware is through web pages. Firstly, because of the popularity of the Web, and secondly, because an attack does not require many prerequisites, only a vulnerable browser. The malicious URL can be sent to the victims via phishing [4] or spear phishing e-mails [5], or there are so called watering hole attacks [6], where the victims are very likely to visit the malicious page. We will never be able to prevent users from visiting malicious pages, so we must detect them.

1.2 Malicious Web Pages

Today's browsers and their plugins are very complex applications. It is inevitable that after design, implementation and testing of such software there are some security flaws remaining after release. Because thousands of programmers and developers use these systems, there are hundreds of flaws and bugs known by the public. For example, according to the Common Vulnerabilities and Exposures (CVE) database, there are almost 1000 Internet Explorer security vulnerabilities discovered [7]. These vulnerabilities are usually corrected immediately in the next release of the affected browser, but there are devices where the browsers are not patched (legacy systems, computers in educational or healthcare facilities). If a skilful attacker exploits a security flaw, all unpatched browsers are potential victims. But even if a browser is updated regularly it is still vulnerable for zero-day exploits. These exploits use bugs which are unknown to the public and only known by the attackers.

The exploits are usually written in script languages which are placed in the HTML content of the websites between `<script>` tags. The common script languages are JavaScript and VBScript (Visual Basic Script), but VBScript is only supported by Internet Explorer and nowadays JavaScript is more significant, so I only concentrated on the JavaScript language (besides HTML). JavaScript can be used for multiple attack types which can be interaction based, or automatic. Interaction based attacks can be placing multiple pop-up windows (adware, malwartising) so the user needs to close them and if they click on the wrong closing button, a file download or redirection will occur. Also, stealing user passwords when entered

in a phishing site, or downloading something if the user clicks on a button, for example, are interaction based. What I am most interested in is automatic attacks (also called drive-by-download attacks). Steps of a drive-by-download attack are usually the following [1]:

1. Victim visits a compromised website with unpatched browser.
2. The compromised website has an HTML element with a source attribute (`<iframe>`, `<embed>` etc.). The browser loads the source, which is usually another website.
3. The page redirects to another page which contains the exploitation script.
4. The browser runs the malicious script, which downloads and runs a malware from another server or does whatever the attacker wants.

1.3 Problem Statement

There are two techniques to detect malicious web pages, dynamic analysis and static analysis.

Dynamic analysis uses isolated environments, such as sandboxes and virtual machines, which are also called high- or low-interaction honeyclients. Their task is to visit suspicious web pages with full-featured browsers, load and execute every script on the page and analyse what has happened after the execution. To decide whether the page is malicious or not, mainly downloaded files, temporary folders and started processes are checked. One of the advantages of a dynamic analyser is that it can deal with highly obfuscated JavaScript code. To evade detection by a rule-based or regular expression based anti-malware software attackers usually obfuscate the code, so the actual exploitation script is hidden under several layers of obfuscation. A dynamic analyser by instrumenting a JavaScript engine can deobfuscate the code and analyse the hidden part. However, the main problem with this technique is the performance. After the execution and analysis - which also requires some time - the virtual machines have to be terminated, restored to a checkpoint and started again which takes several minutes. For example, an IT security company called Ukatemi Technologies[3] (cofounded my consultant Dr. Levente Buttyan) is able to analyse a few thousand URLs, but they receive around a million potentially malicious URLs daily.

[3] `http://www.ukatemi.com/`

Furthermore, this overhead makes it impossible to use dynamic analysis transparently and sensibly in web browsers.

Static analysis is a much faster but less precise technique. It is based on machine learning and tries to classify the web pages by analysing their lexical and syntactical parameters (usually called as features). This method does not need an isolated environment or a fully functional web browser; it only has to download the resources via HTTP(S) without executing the scripts or creating the HTML DOM tree. As a result, it won't give sophisticated results, only the probability of the page being malicious or only the predicted class, but it can analyse around 100 times more pages as dynamic analysis. A static analyser can be used as a prefilter for a dynamic analyser, or it can be integrated in a browser as a filter to warn the user about possible threats.

There were several publications about static analysis around 2010 (see Section 2 on the state of art), but in the last few years no important results were published and there are no open source implementations. My goal is to create a static analyser filter based on the previous results and findings.

1.4 Overview of Approach

In this paragraph, I briefly go through the main steps of my approach, for a more detailed discussion, the reader is referred to Section 3 of this document.

1. First of all, I read and studied research publications available on the internet. These publications were written between 2007 and 2012, and since then, no relevant work has been presented at major conferences.
2. As the static analysis is based on machine learning, therefore, I collected benign and malicious HTML and JavaScript sources for the training set and the validation set. For benign samples, I implemented a web crawler, and crawled popular websites. For malicious samples, I used VirusTotal's[4] notification and alerting system and Ukatemi's malware database.
3. Next, I selected the HTML and JavaScript features for the classifier, and extracted the features from the training dataset.

[4] `https://www.virustotal.com`

4. I implemented a Naïve Bayesian classifier for the filter and trained it with the extracted features. I also trained and evaluated multiple different classifiers using the Weka [8] machine learning toolkit.
5. I validated my Naïve Bayesian classifier with a validation dataset to see the False Positive and False Negative Rates. I compared the results to other classifiers in Weka. After seeing the results, I also evaluated a Random Forest classifier for the filter.
6. Finally, I have made some performance measurements on the prototype of the filter created from three parts, the web crawler, the feature extractor, and the classifier.

1.5 Main Results

We focused on the performance and the accuracy of the filter.

After a test run, the filter was able to analyse around 100.000 URLs in 3 hours, which means the filter can analyse almost a million URLs per day, which is very good for using it as a prefilter for a dynamic analyser.

The False Negative Rate of the filter, using my Naïve Bayesian classifier, was around 20-30% depending on a threshold parameter. Only 3 classifiers from Weka could reach this result. The Random Forest was the best classifier with around 98% Precision and 80% Recall, therefore, I also evaluated a Random Forest classifier for the filter, which produced the same results as the one in Weka. The accuracy of the classifiers was 5% better using JavaScript features than using HTML features.

1.6 Paper Organization

The rest of the paper is organized as follows. Section *2 Related Work* gives some information about publications, and research results I used during my work. Section *3 Our Approach* gives a brief inspection about the theoretical part of my approach to the task, how I chose the features, and how the classification works. Section *4 Implementation* describes every important implementation detail including web crawling and feature extraction. It also states the technologies I used. Section *5 Results* is about the performance and the quality of the filter. Section *6 Future Work* describes the possible improvements in the performance and the quality. Section 7 concludes, and finally, there are the acknowledgements, the references, and the appendix at the end of the paper.

2 Related Work

This section is about technologies and research findings on which my work is based.

2.1 Dynamic Analysis

Nozzle [9][10]: In 2008 Microsoft Research presented an effective technique for detecting heap-spraying[5] attacks only. Through runtime interpretation Nozzle scans heap allocated object data to identify valid x86 code sequences, disassembling the code and building a control flow graph. To intercept function calls that allocate and free memory, they used Detours [11], Microsoft's software package for re-routing Win32 APIs underneath applications. With optimizing detection threshold parameters they were able to produce no false positives and no false negatives on their samples. However, because of runtime analysis, Nozzle increased execution overhead with $10 - 250\%$ per web site, depending on the sensitivity.

Monkey-Spider [12]: Monkey-Spider is a malicious web site detector system with low-interaction honeyclients. Low-interaction honeyclients are not real systems. They emulate real systems and services, typically on virtual machines. They use a mail spamtrap to generate seeds for the Heritrix web crawler[6]. The web crawler downloads content from the World Wide Web. In the next step, they analyse the downloaded content with different anti-virus solutions and malware analysis tools like ClamAV[7], Avast [8]and CWSandbox[9]. With multiple detection software, they were able to produce low True Positive and False Negative Rates, but the drawback of their approach is the slow content analysis. Signature-based detection is fast but cannot detect new attacks. Behaviour-based scanning in sandboxes is more precise but takes a longer amount of time.

---

[5] In heap-spraying attacks, the attacker attempts to inject code somewhere in the address space of the target program, and through a memory corruption exploit, coerce the program to jump to that code. It can be done with JavaScript, exploiting a browser vulnerability [9].

[6] https://webarchive.jira.com/wiki/spaces/Heritrix/overview

[7] https://www.clamav.net/

[8] https://www.avast.com/index

[9] http://cwsandbox.org/

ITWEF: ITWEF is a low-interaction honeyclient developed and maintained by Ukatemi Technologies. It uses around 10-20 virtual machines to visit web pages. The operating systems on the virtual machines are Windows 7, Windows 8.1 and Windows 10. The virtual machines are using Internet Explorer to visit potentially malicious web pages, which are provided by web crawlers: MSICrawler, QutteraCrawler [13] and MRGCrawler. After loading the page and executing the JavaScript code, the suspicious processes and created and downloaded files are analysed. ITWEF also dumps and saves the network traffic from and to the virtual machines and provides the communication in *.saz* files used by the Fiddler network analyser tool[10]. The problem with ITWEF is that it can only analyse a few thousand URLs per day, but the crawlers provide around a million URLs. My filter is planned to be a prefilter for ITWEF.

JSAND (and Wepawet) [13]: Cova *et al.* presented a classifier based on static and dynamic features. The system visits web pages with a customized browser (they have instrumented Mozilla's Rhino interpreted to extract dynamic features), which loads the page, executes its dynamic content, and records the events used by their anomaly detection system. They were able to reduce False Positive and False Negative Rates below 1%. JSAND was made available as part of an online service called Wepawet[11], where users can submit URLs and files that are automatically analysed, delivering detailed reports about the type of observed attacks and the targeted vulnerabilities. Unfortunately Wepawet has been shut down due to maintenance and resource problems, and the creators have recently founded an IT security company called Lastline[12].

Caffeine Monkey [15]: Ben Feinstein and Daniel Pack from SecureWorks created the Caffeine Monkey system. The core of the system is a JavaScript engine based on extensions to the open source SpiderMonkey[13] JavaScript implementation engine from Mozilla. After a Heritrix web crawl, each collected JavaScript sample was run through the Caffeine Monkey engine. The engine hooks a number of interesting functions, like `eval, escape,`

[10] https://www.telerik.com/fiddler

[11] http://wepawet.cs.ucsb.edu/

[12] https://www.lastline.com/

[13] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey

`document.write`. The goal of their work was to deobfuscate samples correctly, so they didn't present results about performance or detection rate.

Zozzle [10][16]: Zozzle is a JavaScript malware detector from Microsoft Research. They call their approach "mostly static", because it is much faster than Nozzle [9] and it is based on static features. However, to deal with obfuscation they intercepted calls to the `Compile` function of Internet Explorer's JavaScript engine located in `jscript.dll`, which is a dynamic approach. This function is invoked when `eval` is called and whenever a new code is included with an `<iframe>` or `<script>` tag. Once they had the deobfuscated code, they built its Abstract Syntax Tree (AST). They extracted specific features from the AST (expressions and variable declarations), but they were only interested in the presence or the absence of the feature in the code. They included only those features in the classification whose presence was correlated with the categorization of the script (benign or malicious). They used a Naïve Bayesian classifier. With around 400 automatically selected features, they were able to produce $< 1\%$ False Positive Rates, and $\sim 7\%$ False Negative Rates. For the majority of files, classification could be performed in under $4ms$.

There are several other dynamic technologies to which the above mentioned researches referred. Just to pick a few: HoneyMonkey by Microsoft Research [17], SiteAdvisor[14] by McAfee, MITRE Honeyclient [15]by MITRE and PhoneyC by Jose Nazario [18].

To sum up, dynamic analysis always executes the JavaScript code on a page in some way. Usually does it on an isolated environment to see the result of the execution, or to deobfuscate the code. Executing JavaScript is a time-precision trade-off, it makes the analysis slower, but more precise. Although, I created a fully static analyser, the methods and techniques used in the dynamic approach were useful, and will be useful in the future of my project.

[14] `https://home.mcafee.com/root/landingpage.aspx?lpname=get-it-now&affid=0&cid=170789`

[15] `https://www.mitre.org/research/technology-transfer/technology-licensing/honeyclient`

2.2 Static Analysis

P. Likarish, E. Jung, and I. Jo [19]: Likarish *et al.* published an approach to detect obfuscated malicious JavaScript with handpicked static features. They used the normalized frequency of each JavaScript keyword as a feature and they also defined features to describe the script's visual appearance (e.g., number of lines, number of Unicode symbols, average string length). They trained multiple classifiers with benign and manually reviewed malicious samples. The Naïve Bayesian classifier was the least precise with ~80% Precision[16], ~66% Recall[17] and ~99% Negative Predictive Power[18].

Prophiler [20]: Canali *et al.* presents a lightweight static filter, called Prophiler. They combine HTML-, JavaScript-, and URL-based features (e.g., number of specific tags, or keywords) to train multiple classifiers. The filter is fully static; it does not execute the JavaScript code to deobfuscate it, and it uses special features to detect obfuscation. They were able to discard benign pages easily, False Positive Rates were around 2% with most of the classifiers, but the average of the False Negative Rates were around $20 - 30\%$. They used the filter as a prefilter for Wepawet.

2.3 Difference from Our Approach

Our approach is similar to the Prophiler and the approach by P. Likarish *et. al.* and is based on them. I created a fully static analyser with web crawling, feature extraction and classification.

To collect benign files I created my own crawler in C#, which is more customizable and scalable than to use Heritrix for example. I used this program in the filter too. None of the above mentioned approaches used their own crawler, they mainly used Heritrix. I collected the malicious files from VirusTotal and Ukatemi's malware database, which is a different source and not used in the other approaches. The features I used are the subset of the features used in the previously mentioned static analyzers. For classification I tried the Weka software too, but I also implemented my own Naïve Bayesian classifier, and I also used a Random Forest implementation for the filter. The complete implementation process is my own solution and it contains my own ideas.

[16] The ratio of (malicious scripts classified correctly) / (all scripts classified as malicious).

[17] The ratio of (malicious scripts classified correctly) / (all malicious scripts).

[18] The ratio of (benign scripts classified correctly) / (all benign scripts).

3 Our Approach

This section provides a detailed overview on how I used machine learning techniques to distinguish between malicious and benign web pages.

One of the main usages of machine learning is classification. The purpose of classification is to divide unseen inputs into previously determined classes. The core of a classifier is the model. Creating a classifier requires three phases. The first phase is feature selection. To achieve a high detection rate, those features should be selected which describe the difference between the classes the best. The second phase is the training, where the classifier creates the model from a provided and labelled dataset (supervised learning). The last phase is the evaluation, or detection phase, where the classifier tries to classify previously unseen inputs using the model. The evaluation phase usually starts with a validation of the classifier. The purpose of the validation is to estimate how well the classifier will perform on real-world data, so the classifier is evaluated on previously unseen, but labelled dataset [21].

3.1 Feature Selection

Selecting features is the crucial part of the classification. We should find and use those features, which describe the difference between a malicious and benign file the best. The following features are static features, which can be extracted from the files without running them. The extraction process only requires parsers for HTML and JavaScript, but there are features which can be extracted only by interpreting the file as a usual text file and analysing its raw content.

The following features are commonly used as HTML and JavaScript features. All of them are used by the Prophiler [20], or Zozzle [16], or JSAND [13], or Likarish *et al.*[19], therefore, I also decided to use these features. Creating new features would be interesting if we modify the filter to detect an exact type of malware.

There are three types of features: discrete (the number of artefacts), continuous (average, percentage of values), or logical (presence or absence of an artefact)

3.1.1 HTML Features

I used the following 16 HTML features [20]. A chosen feature does not mean that the connected tag or attribute is only used for malicious purposes; it means that it can be used, so it is important to take into account.

Discrete features:

Number of `<iframe>` *tags*: *Iframe* tags can be used to insert contents from another website within a web page as if they were part of the current page. In an *iframe* attack, the hacker embeds a malicious iframe code snippet in one's website page. When anyone visits that page, the hidden *iframe* code secretly downloads and installs a malware [24].

Number of hidden elements: The *hidden* attribute can be used, to visibly hide an element from the user. It is a common technique to hide a button for example, so the user does not recognize it and accidentally clicks on it.

Number of small elements: Setting *width* and *height* attributes of an element very small (1-2 pixels) is also a common technique to hide a malicious element. I consider a tag small if the area is smaller than or equal to 30 pixels, or one of its sides is shorter than or equal to 2 pixels [20].

Number of `<script>` *tags*: *Script* tags are to insert inline or to reference other scripts which can either be malicious or benign.

Number of `<embed>` *tags*: *Embed* tags can reference a malicious source in the *src* attribute. It also can be used in a cross-site scripting attack [25].

Number of `<object>` *tags*: Same as *embed*, but it has an *object* attribute.

Number of sources from an external domain: If a benign site is compromised, it is usual that the attacker just places an element with a source attribute pointing to the actual malicious site on an external domain.

Number of parsing errors during parsing: I used an HTML parser which always parses the given document without exception, but provides a list of errors while parsing (e.g., tag is not closed). A benign site is likely to produce fewer errors.

Number of elements in the wrong place: I checked the places of *script, object, embed, frame, iframe* and *form* according to the allowed positioning in the HTML 4 DTD [26]. In a compromised page it is common to see these elements in strange positions.

Number of included URLs: Including multiple URLs can mean that the site is popular, and not likely to contain a malicious code.

Number of characters in the HTML document: The number of characters in the HTML files is expected to be smaller in malicious pages, which are not the results of a corruption of a web page. These pages are not focusing on the good user experience and just contain what is necessary for the attack.

Continuous features:

Percentage of JavaScript content: Obfuscated scripts are usually large and contain huge strings, which could mean higher percentage.

Other continuous features: *percentage of whitespace in the HTML document.*

Logical features:

Presence of `<meta http-equiv=refresh>` *tag*: This tag can be used to refresh the page in a given time, but can also be used as a method of URL redirection [27].

Presence of scripts with wrong extension: A referenced JavaScript file in a source tag normally has a correct *.js* extension. A file without an extension or with a wrong extension might be malicious.

Presence of double documents: In the HTML specification it is not allowed for a page to contain multiple *html, head, body* or *title* tags, but this can be seen in some malicious web pages as a side effect [20].

The number of characters and the percentage of whitespace features can be calculated without parsing the HTML. For the other features, the HTML document has to be parsed.

3.1.2 JavaScript Features

I used the following 22 JavaScript features [19][20]. Most of the following features require parsing the JavaScript file. After parsing the source, I traversed the nodes of the AST provided by the parser to collect the features (for detailed information the reader is referred to Section 4.2.2).

Discrete features:

Number of eval calls: The most popular way of direct dynamic code evaluation is through *eval()* calls. After deofuscation usually an *eval()* is called to execute the deobfuscated JavaScript code [29][30][32][33].

Number of timer function calls: Using *setInterval()*, and *setTimeout()* to periodically do something (e.g., refresh the page) can be used for malicious purposes [34].

Number of string modification function calls: To deobfuscate multiple layers of JavaScript, string modification functions are used to split a long string and to generate a new correct JavaScript code [33]. I focused on the following string modification functions: *slice()*, *subStr()*, *substring()*, *replace()*, *concat()*, *charAt()*, *charCodeAt()*, *split()*.

Number of navigator properties used: It is common in malicious scripts, to identify the browser used by the victim, or to get other information about the victim's environment to use the properties of the *navigator* object [32]. I focused on the following properties: *appCodeName*, *appName*, *appVersion*, *cookieEnabled*, *geolocation*, *language*, *onLine*, *platform*, *product*, *userAgent*.

Number of DOM modification function calls: JavaScript can modify the HTML Document Object Model (DOM[19]) tree in runtime, with DOM modification function, and instantiate vulnerable components. It is possible to place new, and delete or modify already existing elements, which can be used for malicious purposes [29][32][33]. I searched for the following function calls: *getElementById()*, *getElementsByTagName()*, *getElementsByClassName()*, *createElement()*, *removeChild()*, *appendChild()*, *replaceChild()*, *write*, *clearAttributes()*, *insertAdjacentEmlement()*, *replaceNode()*.

Number of long string: An obfuscated JavaScript usually contains a few long strings (even thousands of characters long). These strings are deobfuscated with string operations (*split,slice*), to get the real script [30][32][33]. I call a string long if its length is above a certain threshold; I used 40 characters for the threshold [20].

Number of long variable or function names: Malicious codes usually have long randomly created function and variable names. On the other side, a benign code is not likely to contain long names, because it is not easily readable. I call a name long if its length is above 20 characters.

Number of suspicious strings: A code, which contains strings with *shell, spray, evil,* and *crypt* in them are likely to be malicious. I search for the occurrences of these four words.

Number of strings containing iframe: Creating an *iframe* tag dynamically with JavaScript is a common method to place malicious references to the HTML code [29]. It can

[19] `https://www.w3schools.com/js/js_htmldom.asp`

be done like this: `document.write("<iframe src="malicious URL"></iframe>").`

Number of strings containing suspicious tag names: Besides *iframe*, other HTML elements can be placed while executing a JavaScript code. I test the presence of *frame, object, embed*, and *script* words as a part of a string in the code.

Number of Unicode characters: A shellcode can contain multiple Unicode characters. A shellcode is a piece of malicious code, which is placed in the browser's or the plugin's memory, and executed through exploiting a buffer overflow.

Other discrete features: *number of method calls*, *number of strings*, *length of the script*, *number of lines, maximum length of strings.*

Continuous features:

Entropy based features*: average entropy of strings, max entropy of strings, entropy of script*: These features are used to describe the randomness of the strings and the script. If a script contains an obfuscated code, it is likely that the entropy will be higher.

Other continuous features: *average string length, average line length* and *percentage of whitespace.*

The *number of lines*, the *length of script*, the *average string length*, the *maximum length of string*, the *average line length*, and the *percentage of whitespace* features can be calculated without parsing the JavaScript; the others require parsing, and traversing the AST.

3.2 Training

The purpose of training is to acquire the characteristics of a dataset, which will be referred to as the training set. Every element of the training set contains multiple features and the name of the class (a label) in which the actual element belongs. The classifier tries to derive a model from the given features and the classes they belong.

In the context of web pages, the classification of every element of the training set is a list of features extracted from the HTML, or JavaScript code of the web page, and a *benign*, or *malicious* label representing the class of the element. To create a precise classifier, the training set has to be correctly labelled manually and also, the features have to be carefully chosen to describe the differences between malicious and benign codes (for more information see Section 3.2).

To train a classifier, the first task is to collect HTML and JavaScript resources which are likely to be benign or likely to be malicious. Collecting malicious files and collecting benign files requires to different methods.

Benign training set: The most common method for collecting benign samples is with a web crawler. I tried out a few web crawlers and crawler frameworks, like Heritrix – the Web Archive's crawler, Scrapy [20]– a web crawling framework in Python, but I decided to implement my own web crawler in C#. The main reason was that implementing the crawler myself, customizing it will be easier (like setting HTTP headers, parsing the HTML or JavaScript). Furthermore, I can use the crawler not only for gathering the training set, but also in the filter for crawling potentially malicious web sites. I collected 439 URLs of the most popular web pages according to Alexa.com[21], Quanticast[22], Wikipedia and other sites. These URLs provided the benign seed for my crawler. My assumption was that these websites and the sites which they reference are not likely to contain malicious code. My implementation makes it possible to start multiple crawlers simultaneously; one crawler is responsible for crawling one seed URL. The phases of the crawling can be seen in Figure 3.1. For implementation details (handling HTTP messages, parsing the HTML) the reader is referred to Section 4.

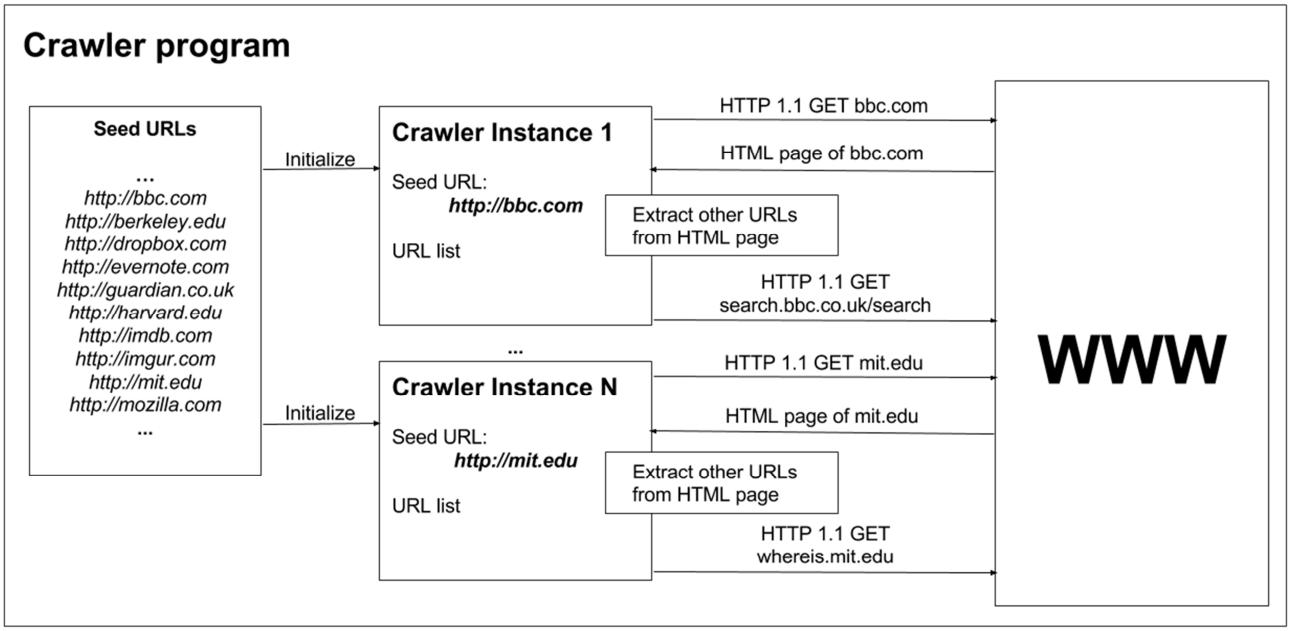

Figure 3.1. The phases of crawling

[20] https://scrapy.org/

[21] https://www.alexa.com/topsites

[22] https://www.quantcast.com/top-sites/

I focused only on HTML and JavaScript resources and therefore used the following algorithm to determine the type and the examining process:

1. HTTP `Content-Type` header of the response: I accept `application/xhtml+xml`, `text/html` and `text/plain` `MIME-types` as HTML and `application/x-javascript`, `application-javascript` and `text/javascript` `MIME-Types` as JavaScript [23].
2. If the `Content-Type` header is set but not to an accepted `MIME-type`, then I discard that resource (it could be a picture, flash etc.).
3. If the `Content-Type` header is empty, I try the following:
    a. If the resource contains an `<html>` tag, I interpret it as an HTML document.
    b. If it does not contain an `<html>` tag, and the JavaScript parser can parse it, I interpret it as a JavaScript file.
    c. Otherwise I discard the resource.
4. If the resource is HTML, I extract new URLs, and the JavaScript code on the page.
5. I save both the HTML and the JavaScript source.

To get other URLs from the seed web page, I parsed the HTML file. Table 3.1 shows the HTML tags, and the attributes of the tags which I focused on to extract resource URLs. After collecting every URL from the actual page I filtered the URLs. I discarded those URLs which were irrelevant based on their extensions, like `.png`, `.css`, `.swf` etc. These filtered URLs provided the new URLs to fetch.

HTML Tag	*Attribute*
script	src
iframe	src
frame	src
embed	src
object	data
form	action
link	href
a	href

Table 3.1. HTML tags and their attributes used to gather new URLs.

To get the JavaScript code from the HTML file, I examined the `<script>` tags in it. The content of the tag is collected and interpreted as JavaScript if the following conditions stay:

- The tag does not have a `language` attribute or it has a `language` attribute and it is JavaScript, and
- The tag does not have a `type` attribute or it has a `type` attribute and it is `application/javascript`, `application/ecmascript`, `text/javascript`, `text/ecmascript` [23].

All of the inline JavaScript codes of an HTML page are merged together and treated as one script file. This was necessary because most features of the individual inline scripts are meaningless if the code is a few lines long. Furthermore, the browsers are interpreting the JavaScript code of a page as a whole.

With the above mentioned technique I was able to gather and save the HTML and JavaScript files while crawling the seed URL. To remove duplicate files, I matched the SHA-256 hashes and the actual URLs.

I started a web crawling process with the following parameters: 439 seed URLs in a file, 2 crawlers working in parallel on a single seed URL each, every crawler downloads at least 300 other URLs, every crawler sends max 20 requests simultaneously. The complete process took around 12 hours, during which 107464 distinct HTML and 92217 distinct JavaScript files were collected and saved (see Table 3.2), which is 5 $file/sec$. The crawling process was run on a Ubuntu virtual machine on a VMWare ESXi server. It had limited resources because of the other virtual machines on the ESXi server. In the future, on a stronger environment, the performance can be optimized. The bottleneck of the process is fetching the URLs, which can take seconds, so increasing the parallel requests can fasten up the process, but the number of threadpool threads should be considered too (see Section 4).

Malicious training set: Creating a malicious training set is hard to do by crawling the web. The problem is to find URLs which are very likely to contain malicious code. There are multiple blacklists available on the web like MalwareDomains[23], but these lists are frequently changing because the attackers are also monitoring these blacklist and they usually terminate

[23] `www.malwaredomains.com/.`

the page if it appears on a blacklist. Ukatemi has a VirusTotal licence with which the following is possible: we can use VirusTotal's messaging service by specifying a Yara[24] rule, that is matched by VirusTotal to every freshly uploaded sample. We created a rule, which usually matches on JavaScript files (see the exact rule in the Appendix). By also providing an email address, VirusTotal sends a message every time someone uploads a file to its server which matches the specified Yara rule. We did not provide a rule for HTML files because our experience was that it is very likely that the uploader uploads an HTML file which contains a malicious JavaScript code rather than only uploading the JavaScript source. The message sent by VirusTotal is plain text, but it has a well-defined structure. It contains information about the uploaded file like MD5 and SHA-256 hashes, file type and the list of alerts of antivirus products triggered by the file. For example, we found in this way Ramnit Trojans, Facelikers, FakeJQuery Trojans, LikeJacks, Phishing malwares, Adwares, Iframe Trojans, Redirectors and Clicker Trojans.

During a month's time, I received around 70000 emails. I wrote a program in C# to download and parse the messages. After the parsing, I checked whether the referenced file caused any alert on any antivirus. If not, I discarded the message. If yes, I gathered the SHA-256 value from the message, and I used Ukatemi's malware database to collect the actual file from the hash. They provide a REST service, so if one sends the hash of a file, you receive the file if it is in the database. If the file was HTML, I extracted the JavaScript code, like from the benign files. From the 70000 messages, most did not contain any alerts, some were duplicates, some of them were not JavaScript or HTML (usually ELF or ZIP), and some of them were not in the database. In the end I had 1594 HTML and 1784 JavaScript files assumed as malicious (see Table 3.2). I only examined the subset of the files and not all, which could cause some problems (for more information see Section 6).

	HTML	JavaScript
Benign	107464	92217
Malicious	1594	1784

Table 3.2. Number of collected samples

After collecting the training set, feature extraction comes from the collected samples (detailed in Section 4). Finally, the classifier creates the model from the provided training set.

[24] Yara is a popular pattern matching tool used widely by identifying and classifying malware. http://yararules.com/

The actual method to create the model depends on the classification algorithm described in Section 3.1.3.

3.3 Evaluation

Validation: To validate a classifier, a validation dataset is used, which is labelled just like the training set, but it is not used in the training phase, which means that these values are unseen for the classifier. The purpose of the validation is to estimate the real world performance of the classifier. To measure the performance and precision of a classifier, the confusion matrix (Table 3.3) and other attributes [22] are calculated:

JavaScript/HTML	**Labeled as Malicious**	**Labeled as Benign**
Classified as Malicious	**True positive (TP)**	**False positive (FP) Type I Error**
Classified as Benign	**False negative (FN) Type II Error**	**True negative (TN)**

Table 3.3. Confusion matrix for web page classification (template source: https://en.wikipedia.org/wiki/Confusion_matrix)

- *False Positive Rate (FPR)*: $FP/(FP+TN)$ The probability of labelling a benign script malicious. If high, it causes an overhead for the underlying dynamic analyser.
- *False Negative Rate (FNR)*: $FN/(FN+TP)$ The probability of labelling a malicious script benign. If high, we miss analysing some malicious scripts; also we allow the user to visit the web page.
- *True Negative Rate (TNR), Specificity*: $TN/(TN+FP)$ The probability of classifying a benign script correctly (as benign). Equals $1-FPR$.
- *True Positive Rate (TPR), Recall, Sensitivity*: $TP/(TP+FN)$ The probability of classifying a malicious script correctly (as malicious). Equals $1-FNR$.
- *Positive Predictive Value (PPV), Precision*: $TP/(TP+FP)$ The probability of the correct classification if the classification result is malicious. If high, a maliciously labelled script is very likely to be malicious.

- *Negative Predictive Value (NPV)*: $TN/(TN+FN)$ The probability of the correct classification if the classification result is benign. If high, a benignly labelled script is very likely to be benign.
- *F-measure*: The F-measure is the harmonic mean of Precision and Recall. It can be calculated with the following formula (3.1):

$$Fmeasure = 2 * \frac{Precision * Recall}{Precision + Recall} \quad (3.1)$$

During my work, I focused mainly on false positive and False Negative Rate. It is important to achieve low False Negative Rate, otherwise the classifier will label a malicious script benign. If we use the filter as a prefilter, the dynamic analyser will not analyse those scripts, and if we integrate it in a browser, the filter will not notify the user about the possible threats. Low False Positive Rate can also be necessary, otherwise the false positive samples will cause an overhead for the dynamic analyser, and also a user may be annoyed by the usual alerts, and will more likely disable the filter. These two parameters are in an inverse relationship: we can lower the FNR by increasing the FPR and vice versa.

I used two different types of validation techniques during my work. For my implementation of the Naïve Bayesian classifier and for training the Random Forest, I divided the training set to 70%-30%. With the 70% I trained the classifiers, and I validated them with the rest. The other method I used is the n-fold cross validation [28]. A trained and validated multiple other classifiers with the Weka machine learning tool, which provides cross validation. I used the most common 10-fold cross validation, besides 70-30% partitioning. In 10-fold cross validation, we separate the whole dataset into 10 equal groups. We train the classifier with $9/10^{th}$ of the data and validate it with the remaining $1/10^{th}$. Finally, we calculate the average of the above mentioned parameters.

Evaluation on real-world data: During the creation of this paper I was not able to create exact measurements about the precision of the filter on real world data. To measure the detection rate of the filter would require ITWEF to analyse the same URL list twice, with and without the filter which is not trivial. Also, optimizing the performance, thread numbers and parallel requests are yet to be done.

3.4 Classifiers

Choosing the classifier plays an important part in my work. Choosing a more complex and robust classifier can produce lower False Positive and False Negative Rate, however, the implementation of the algorithm is much harder. Choosing a simple classifier usually means higher FPR and FNR, but the implementation is easier and we can concentrate on the results sooner.

I chose to implement a Naïve Bayesian classifier in the first place, which is one of the simpler classifiers. Besides the easy and fast implementation, the related works showed that the precision of a Naïve Bayesian classifier can be unexpectedly high, reaching other more sophisticated classifiers in quality. Zozzle uses only a Naïve Bayesian classifier and also Prophiler and the approach by Likarish *et al.* presented results with it.

However, I tried the Weka machine learning platform as well – which implements multiple classification algorithms, and provides a user friendly interface to import a feature dataset and to run the classification – to analyse the quality of other classifiers and to compare it with my implementation.

After comparing my Naïve Bayesian Classifier, with the classifiers from Weka, the results showed that the Random Forest algorithm was the best in Precision, Recall and F-measure. Therefore, I used and evaluated a Random Forest classifier implementation for the filter.

3.4.1 Naïve Bayesian Classifier

Bayesian classifiers are statistical classifiers. They can predict the probability that a given sample belongs to a particular class. In my case, a sample is a tuple of feature values (16 HTML and 22 JavaScript), the classes are *malicious* and *benign*. Naive Bayesian classifiers assume that the effect of an attribute value on a given class is independent of the values of the other attributes. This assumption is called class conditional independence. While this assumption is incorrect (e.g. string-based features, entropy-based features, function call-based features), the classifier yields good results [16][40]. Another assumption is that the distribution of each feature in the training dataset is approximately the same as in the real world samples.

Bayesian classifier is based on Bayes' theorem. According to the theorem, the probability of a sample belonging to a class C_i, with $F_1, F_2, \dots, F_n$ features can be calculated with the following formula (3.2):

$$P(C_i|F_1, F_2, \dots, F_n) = \frac{P(C_i)P(F_1,F_2,\dots,F_n|C_i)}{P(F_1,F_2,\dots,F_n)} \qquad (3.2)$$

The denominator does not depend on class, so only the nominator should be calculated. This will give a likelihood value, not an actual probability. The nominator can be transformed using the rule of conditional probability (3.3):

$$P(C_i)P(F_1, F_2, \dots, F_n|C_i) = P(C_i) \prod_{k=1}^{n} P(F_k|F_{k-1}, \dots, F_1, C_i) \qquad (3.3)$$

Then, by applying the independence assumption, we gain the following equation (3.4):

$$P(C_i) \prod_{k=1}^{n} P(F_k|F_{k-1}, \dots, F_1, C_i) = P(C_i) \prod_{k=1}^{n} P(F_k|C_i) \qquad (3.4)$$

$$P(C_i) \prod_{k=1}^{n} P(F_k|C_i) \qquad (3.5)$$

We want to calculate (3.5).

The $P(F_k|C_i)$ probabilities can be calculated from the training set. There are three possibilities regarding the type of the feature:

I. If the feature is continuous:
   a) The estimated probability density function can be used to calculate the conditional probabilities.
   b) The feature values can be transformed to ordinal categories using a binning method.
II. If the feature is discrete:
   a) The estimated probability density function, or probability mass function can be used to calculate the conditional probabilities.
   b) The feature values can be transformed to ordinal categories using a binning method.
III. If the feature is logical, the categories are already provided.

I chose to transform every numerical feature into ordinal categories. Some feature's probability density function could be estimated by kernel function. It is a possibility to try in the future, but it is not trivial to use. Most of the features could be intuitively split into three or two categories (e.g. the presence or absence if *iframe*, or *object* tags) after analysing the

histograms, but I chose a more common way to bin the values[25]. For every feature, the mean and the standard deviation were calculated (using SPSS) and the borders of the bins came from the result: $mean - std.deviation;\ mean;\ mean + std.deviation$. So, there were only maximum 4 categories for a feature:

Category I: $x <= mean - std.deviation$

Category II: $mean - std.deviation < x <= mean$

Category III: $mean < x <= mean + std.deviation$

Category IV: $mean + std.deviation < x$

If the lower boundary ($mean - std.deviation$) was undefined (e.g., negative), I used only 3 categories. For logical features I used the *true* and *false* bins as is.

After binning the values I created the frequency tables for each feature, to calculate the conditional probabilities (Table 3.4).

Feature A	Labelled as malicious	Labelled as benign
Category I	M1	B1
Category II	M2	B2
Category III	M3	B3

Table 3.4. A frequency table used to calculate conditional probabilities.

In Table 3.4, M1 is the number of samples labelled as malicious, where the value of Feature A falls into Category I. The other values are calculated like M1. To get conditional probabilities, the values in the frequency table are divided by the number of all maliciously/benignly labelled samples.

From (3.5) we have just calculated the production part. The a priori probability of a class $P(C_i)$ could be calculated from the relative frequency of the classes in the training set, but this would assume that the frequency of the classes in the training set is the same as the probability of a randomly chosen sample being malicious or benign. Therefore, I used the assumption that spam filters usually use [41]: There is no a priori reason for any web page to be malicious rather than benign. So in my calculations: $P(malicious) = P(benign) = 0.5$, so the multiplication with the a priori probabilities can be omitted.

[25] Other common binning (or discretization) methods are *equal width interval*, and *equal frequency interval* discretization, but neither of them could be used for all features.

The classification algorithm for sample S, with feature values $F_1, F_2, \dots, F_n$ will be as follows:

1. Put every feature value in the correct category of the feature.
2. Calculate the frequencies from the corresponding cell of frequency table of each feature, and divide them with the number of malicious/benign samples in the training set. These are the relative frequencies, which by assumption equal the conditional probabilities.
3. Multiply the relative frequencies for both *malicious* and *benign* class separately. The result is: $LH(malicious|F_1, F_2, \dots, F_n), LH(benign|F_1, F_2, \dots, F_n)$, where LH stands for likelihood.
4. Compare the two numbers; the result is the class whose LH value is higher. We can calculate probabilities from likelihoods (3.6).

$$P(malicious|F_1, F_2, \dots, F_n) = \frac{LH(malicious|F_1,F_2,\dots,F_n)}{LH(malicious|F_1,F_2,\dots,F_n)+LH(benign|F_1,F_2,\dots,F_n)} \quad (3.6)$$

5. With the probabilities, it is possible to modify the detection threshold. The classification result is *malicious* if $P(malicious|F_1, F_2, \dots, F_n) > threshold$. With the threshold it is possible to optimize the false positive and False Negative Rate.

3.4.2 Classifiers in Weka

Weka is a collection of machine learning algorithms integrated in one tool. The algorithms can either be applied directly to a dataset or called from a Java code. I used the graphical user interface to import datasets and to run the classification algorithms. I applied the following classifiers on the dataset: Bayes Net [35], Naïve Bayes [36], Logistic [37], J48 [38], Random Forest [39] and Random Tree. The detailed analysis of the classifiers and their algorithms was not in the scope of my work or this paper, so I only present the useful descriptions and publications about them.

Weka produced detailed results about the created models, the TP and FP rate and it presented the confusion matrices, so it was evident to compare the results.

3.4.3 Random Forest

The Random Forest is an ensemble learning algorithm used for classification and regression. An ensemble learning algorithm combines the results of multiple other learning algorithms to improve the accuracy. In case of the Random Forest technique, it creates multiple Decision Trees (the Forest) and uses the combined results of the Decision Trees for classification and regression. Decision Trees are Classification and Regression Trees (CART), because they can be used for classification and regression purposes. I used the trees for classification [44].

The Random Forest algorithm grows the trees as follows [42]:

- Sample the training set at random, but with replacement. This sample will be the training set for growing the tree.
- If there are M features, a number m<<M is specified. At each node, m features are selected at random out of the M and the best split on these m is used to split the node. The value of m is held constant during the forest growing. The common methods to define m are taking the square root of the number of features, or taking the logarithm.
- Common methods to define the best split are using giri index, chi-square, information gain or reduction in variance.
- Each tree is grown to the largest extent possible. There is no pruning. But to avoid large trees, which can cause overfitting, limiting the depth is necessary.
- The prediction of the whole forest comes from the predictions of the trees. In case of classification, the class with the majority of the votes is used.

The advantages of using the Random Forest algorithm relevant for my work are the following: it can handle both numerical and categorical data; it can handle a large dataset and a large number of features; it can give estimates of what variables are important in the classification; also, the generated forests can be saved. However, I have very little control of what the model does, and creating the model can be time consuming [39][42][44].

Optimizing and tuning a Random Forest algorithm for a special purpose is not an intuitive and easy task [43], I did not deal with this during my work. It is a possible way for the future.

4 Implementation

This section describes the implementation process of the filter. It details the training set collection, the feature extraction from the parsed HTML and the JavaScript samples and the process of the classification.

4.1 Programming Environment

The filter was created in the .NET Core 1.1 Framework[26], using the C#[27] programming language. We chose C# because it is a high-level language with thorough documentation. It has language features, like `await-async` for asynchronous programming, *LINQ* for queries and anonymous functions, which are useful for web crawling and feature extraction. Furthermore, thanks to the .NET Core Framework, we can create cross-platform applications with C#. It was necessary for the filter to run on UNIX systems. Although .NET Core was announced only a year ago (June-July 2016) [45], the important packages and libraries for my work are supported by framework. For a code editor the Visual Studio Code[28] was the ideal choice.

I worked on a Ubuntu 14.04 virtual machine, which was running on a VMWare ESXi Server maintained by Ukatemi. Besides development, I used this computer to store the downloaded benign pages and the collected malicious ones. I also used my personal laptop with Windows 10 operating system for development, for analyzing the features.

For version control, I used git[29], which is integrated in Visual Studio Code. For inspecting the extracted features and creating the frequency tables, I used SPSS Statistics[30] from IBM and MATLAB[31] from MathWorks. For running other classification algorithms, I used the Weka machine learning tool[32].

---

[26] https://www.microsoft.com/net/core#windowscmd

[27] https://docs.microsoft.com/en-us/dotnet/csharp/

[28] https://code.visualstudio.com/

[29] https://git-scm.com/

[30] https://www.ibm.com/analytics/us/en/technology/spss/

[31] https://www.mathworks.com/products/matlab.html

[32] https://www.cs.waikato.ac.nz/ml/weka/

4.2 Training

For training the classifiers I had the following tasks to do and implement:

- Gather benign and malicious samples.
- Extract the previously selected feature values from the collected files to create the training set.
- Train the classifier with the training set.

I implemented these functionalities as standalone programs, which can be used individually and also can be integrated into one filter.

4.2.1 Collecting HTML and JavaScript samples

As I described before, it is not trivial to collect malicious files with web crawling. I implemented a web crawler in C# to collect benign samples, and I implemented another application, which downloaded the messages from VirusTotal through HTTP from my mailbox and used the SHA-256 hashes to collect the files from HashDB.

4.2.1.1 Benign sample collection

To collect benign files, I implemented the *Crawler* module as a standalone program, which I also used later for crawling potentially malicious web pages. The program runs through the following steps:

I. ***Initialization***: The *Crawler* module has three command line parameters: the number of parallelly started crawlers, the number of parallel requests in one crawler and the minimum number of URLs to be fetched per crawler. The initial seed URLs can be placed in a text file in the project root, one URL in one line (e.g., *http://facebook.com*). I allowed commenting URLs with '//' for being more user friendly. For each URL, the program starts a separate crawling job, but first, the global HTTP headers are set. For creating and handling HTTP requests and replies I used the `HttpClient` [46] class located in the `System.Net.Http` namespace. I set the following headers and values:

Referrer: https://www.google.com

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8

Accept-Language: en-US,en;q=0.5

Accept-Encoding: gzip, deflate

User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:54.0) Gecko/20100101 Firefox/54.0

These are the traditional headers used by Mozilla and Chrome. This way, I can pretend, that I am requesting the page from a web browser as a regular unsuspecting user.

For using *gzip* and *deflate* decompressions the `AutomaticDecompression` property of the `HttpClient` had to be set using `HttpClientHandler`. To analyse in- and outgoing HTTP messages I also set a proxy for the `HttpClient` to 127.0.0.1:8888 if the HTTP debugging application Fiddler[33] was running on the machine.

II. ***Sending HTTP requests***: After the initialization, the crawling tasks were started for each seed URL (the number of the parallel crawlers was limited by a command line parameter). For containing and maintaining the URLs and other information collected from the seed URL, I created a `Crawler` class which represents one Crawler and is responsible for one seed URL. Each `Crawler` maintains a list about the extracted URLs from the html page located on the seed URL and the other HTML pages linked or referred from the seed.

   The `Crawler` sends multiple HTTP GET requests asynchronously to the extracted URLs (`HttpClient.SendAsync(…)`). C# provides asynchronous programming with `async`, and `await` keywords [47]. After the framework sends a request, it yields the control to the calling function which can continue with the program. This mechanism is useful for I/O-bound tasks (like HTTP requests), and can increase throughput.

   After **any** reply arrives without errors, the `Crawler` processes the response. Technically the `Task` objects returned from the `async` request function are stored in a list. After sending multiple requests, the Crawler waits for any reply to arrive: `await Task.WhenAny(httpRequestTaskList)`.

III. ***Processing the HTTP response***: If the HTTP status code of the result is 200 OK, then it is processed as follows. The `Mime-Type` of the returned resource is determined as detailed in

[33] `http://www.telerik.com/fiddler`

Section 3.2. If the response is HTML or unknown, it is parsed as HTML to collect the referenced URLs[34] and the inline JavaScript code.

IV. ***Parsing the HTML and gathering new URLs and inline JavaScript***: For parsing HTML documents, I used the popular *HTMLAgilityPack*[35], which is available through the NuGet[36] package manager. This library can parse any text (`HtmlDocument.LoadHtml(htmlSource)`) without throwing exceptions, and it provides a list of occurred parsing errors (e.g., `EndTagInvalidHere`, `TagNotOpened`, `TagNotClose`). It also allows for querying and navigating through the created nodes with *XPath*, and *LINQ*. I used both of these technologies. After parsing the file, the presence of the `html` tag is checked. If it is present, then starts the processing of the HTML file.

I iterate through the nodes of the document, and search for specific tags and attributes to gather the URLs. The tags and attributes are already mentioned in Section 3.2, Table 3.1. The values of an attribute can be easily extracted. After collecting the URLs I normalize them: the protocol (`http://`, `https://`) and `www.` are removed. I discard those URLs, which starts with `mailto:`, `android-app:`, `ios-app:` etc., and also those, whose extension is not in a list of accepted extensions: `.js`, `.htm`, `.html`, `.xht`, `.php`, `.asp`, `.aspx`. These are the most common extensions for files which provide or generate an HTML or JavaScript file. Also, the URLs, which were already fetched, are deleted.

V. ***Ending the crawl, saving files***: After reaching the specified number of successfully fetched URLs the `Crawler` saves the collected resources with `.js`, or `.html` extension and exits. A program is waiting for `Crawlers` to finish their task with `Task.WhenAny`, and starts another crawling with a new seed URL.

---

[34] There is no need for parsing the JavaScript while only collecting the training samples. But later, when the Crawler module is integrated with the Feature Extractor module the JavaScript parsing is also started here.

[35] `http://html-agility-pack.net/?z=codeplex`

[36] `https://www.nuget.org/`

The files were saved in a hierarchical structure presented in Figure 4.1.

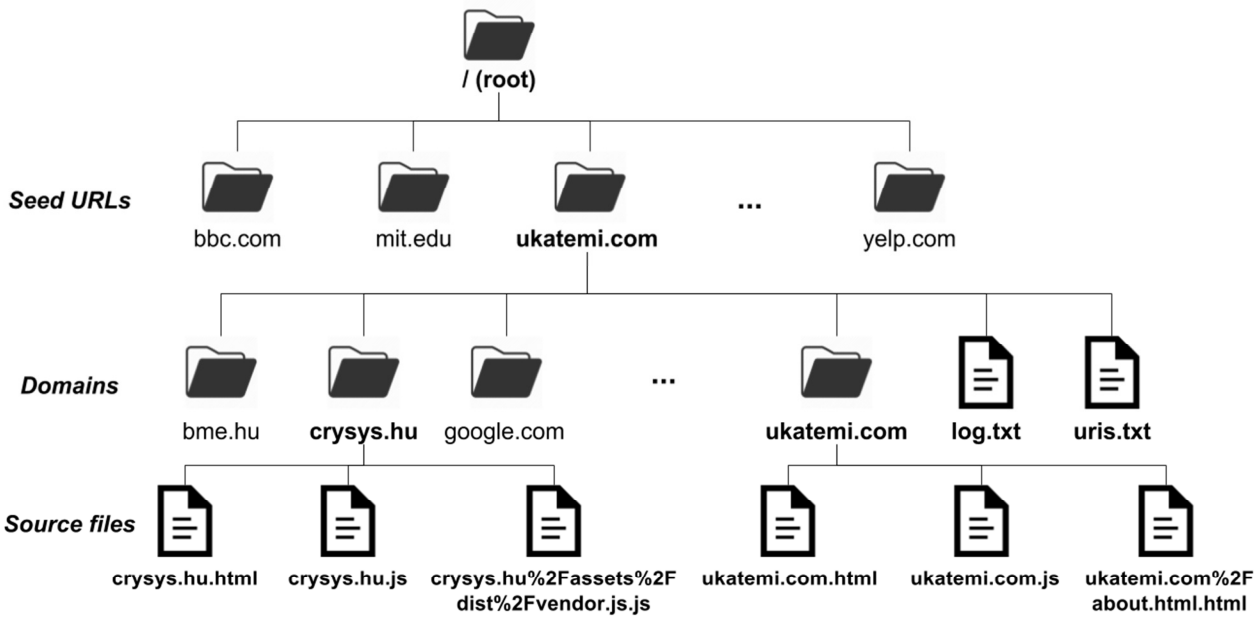

Figure 4.1. The directory structure of the downloaded HTML and JavaScript files.

A directory was created for each seed URL (*ukatemi.com*), and in this, other directories were created for every domain referenced directly or indirectly from the seed (*ukatemi.com, crysys.hu*). The log file for the individual crawl (*log.txt*) and the successfully fetched ULRs (*uris.txt*) by the crawler were placed in the seed URL directory. In every domain directory, the downloaded files were stored with the actual URL as their name[37] (*crysys.hu.html, ukatemi.com.js*). The .html and .js extensions were always concatenated to the end of each filename to easily distinguish between HTML and JavaScript files any time. Later, every file was copied to a single directory and I deleted those, which have the same SHA-256 value, or the same name.

4.2.1.2 Malicious sample collection

As I mentioned above, malicious samples were collected through VirusTotal and Ukatemi's HashDB. The mails from VirusTotal arrived at my *@ukatemi.com* email address. To manually process around 70.000 emails was practically infeasible, so I wrote a program to

[37] The URLs could not be used as file names, because of special characters, so I used the `Uri.EscapeDataString()` from the framework to create the file name.

automatically download the emails, check if they were caught by any antivirus products, and if they were, query HashDB with the SHA-256 hash, and save the actual malicious file.

I connected to the mail server through HTTP. I did not use IMAP because .NET Core 1.1 does not support it (however, it is now supported in the newly released 2.0 version). So I logged in to the web page, opened the mails virtually, fetched the html files and extracted the actual textual messages. An example for a message can be seen below:

```
Link          :
https://www.virustotal.com/intelligence/search/?query=4a745815202dcaeb8229a
9861c43a05d0e66ee8f1cbd4fcb43983911559e18d8
[…]
SHA256        :
4a745815202dcaeb8229a9861c43a05d0e66ee8f1cbd4fcb43983911559e18d8

Type          : HTML
[…]
First country: DE


ALYac                          Trojan.Script.503239
AVG                            JS:Includer-BMA [Trj]
AVware                         Malware.JS.Generic (JS)
[…]
Kaspersky                      Trojan.HTML.Redirector.cv
[…]
ZoneAlarm                      Trojan.HTML.Redirector.cv

2F 68 65 61 64 3E 0D 0A 3C 62 6F 64 79 3E 0D 0A   /head>..<body>..
[…]

EXIF METADATA
=============
[…]
```

I parsed the mail line by line and searched for specific parameter names like *Link, SHA256, Type,* and stored them in a `Map`. After the *First Country* value, I checked whether the antivirus alerts were missing or not. If there were any alerts, I used the HashDB's REST service to download the malicious files by their SHA-256 hash value.

4.2.2 Feature Extraction

The *Feature Extractor* module is responsible for extracting the previously listed features from the downloaded HTML and JavaScript files. The simplifies class diagram of the module is shown in Figure 4.2.

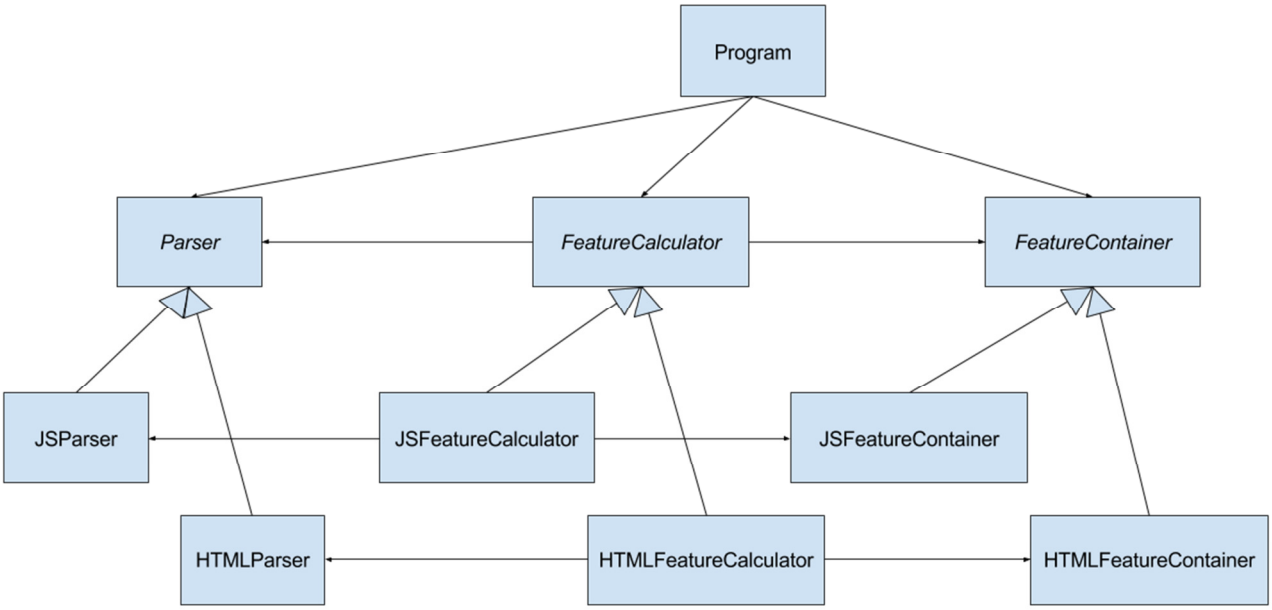

Figure 4.2. The simplified class diagram of the Feature Extractor module.

The module has two tasks. First, it parses the given file, then it extracts (and saves) the features from the parsed file.

I. ***Parsing***: As the *Crawler* module identified the types of the downloaded files and saved them with proper extensions, the *Feature Extractor* module knows exactly how to handle each file. However, the types of the malicious files were not identified during the collection. So, if a file does not have extension, the *Feature Extractor* module tries to parse the file as HTML first, and if the parsed file contains an `html` tag, then it is handled as an HTML file. If it does not contain any `html` tag, then the file is parsed as JavaScript. The JavaScript parser I used throws an `Exception` if the file cannot be parsed as JavaScript. In this case the file is discarded.

The `Parser` class represents the abstract base class of the parsers. The `JSParser` and the `HTMLParser` class inherit from the base class; they are responsible for containing and parsing a file.

I used the same package (*HtmlAgilityPack*) and the same classes to parse a file as HTML as in the *Crawler* module.

To parse a file as JavaScript, I used the *Esprima.NET*[38] package by Sebastien Ros, which is also available through *NuGet*. The package supports the *ECMAScript® 2017 Language Specification*[39], on which JavaScript is based. To parse a file and get the Abstract Syntax Tree the following few lines are needed:

```
Esprima.JavaScriptParse parser =
     new Esprima.JavaScriptParser(content, options);
Esprima.Ast.Program program = parser.ParseProgram();
```

If the file cannot be parsed, the library throws a `ParserException`.

Waiting for the HTTP responses is the most time consuming; it can take 10-30 seconds. However, JavaScript parsing also takes some time. It can take a few hundred milliseconds maximum depending on the size of the file. For one file, it is negligible, but parsing 10000-100000 JavaScript files synchronously on thread can take minutes. Feature extraction is even more time consuming (for detailed results the reader is referred to Section 5.). To reduce this cost I implemented a custom thread pool. It would be easier to use the standard *.NET ThreadPool*[40], with `Task.Run()`[41], but the `HttpClient` also uses the *ThreadPool*, which has limited number of threads in it. My `CustomThreadPool` class is based on this implementation: [48]. It creates custom number of worker threads. They wait for a `Parser` in a `Queue`, which is not parsed the containing file yet. After a new `Parser` arrived, one thread gets it from the queue and starts the parsing. After the parsing has finished, the thread puts the `Parser` to a `BlockingCollection`, which contains the `Parsers` that are ready for feature extraction.

II. ***Feature Extraction***: First of all, I created two enumerations for the HTML and JavaScript features to reference them easily. Extracting features from the parsed HTML files and extracting features from the parser JavaScript files cannot be done in the same way.

HTML feature extraction: The feature extraction from HTML files is done by `HTMLFeatureCalculator` (see Figure 4.2). First, the `Calculator` does a

---

[38] https://github.com/sebastienros/esprima-dotnet.

[39] http://www.ecma-international.org/publications/standards/Ecma-262.htm

[40] https://msdn.microsoft.com/en-us/library/system.threading.threadpool (v=vs.110).aspx

[41] https://msdn.microsoft.com/en-us/library/hh195051(v=vs.110).aspx

preprocessing, it runs through every node in the `HtmlDocument` created by the `HTMLParser`, and collects the following nodes in one `Map` with the element names as keys: `script, iframe, frame, embed, object, form, link, a`. This way, the `Calculator` runs through the nodes only once and not 7 times.

The *number of* `<iframe>` *tags,* the *number of* `<script>` *tags,* the *number of* `<embed>` *tags,* and the *number of* `<object>` *tags* are the sizes of the `Maps`.

The *number of hidden elements* is extracted using `//*[@hidden]` XPath query. XPath strings can be passed to the `document.DocumentNode.SelectNodes(xpath)` method, where the `document` is the parsed HTML.

The *number of small elements* is extracted using `//*[@width<=2 or @height<=2 or @height*@width<=30]` XPath query.

The *number of sources from an external domain* feature is impossible to be implemented because for the malicious files the source URL of the file is unknown, so it cannot be compared to the referenced ULRs in the file. Therefore, I only counted the referenced absolute URLs (starting with *http://* or *https://*). Those nodes, which can have a source attribute are already collected in a `Map`.

The *number of included URLs* is extracted almost the same way as the previous feature, but the relative URLs were counted too.

The *number of parsing errors during parsing* is provided by the parsed document in `document.ParseErrors`, which is an `IEnumerable<HtmlParseErrorCode>`.

To collect the *number of elements in the wrong place* I counted the result of the `/html/head//script|/html/body//iframe|/html/body//frame|/html/body//embed|/html/body//object|/html/body//form` XPath query.

The n*umber of characters in the HTML document*, and the *percentage of whitespace in the HTML document* can be calculated from the unparsed, raw HTML.

For calculating the *percentage of JavaScript content*, the JavaScript content is fetched from the document, which is mentioned in Section 4.2.1.1.

The *presence of* `<meta http-equiv=refresh>` *tag* is calculated from the result of the `//meta[@http-equiv='refresh']` XPath. If it returns `null`, there are no matching nodes.

To find the *presence of scripts with wrong extension* the *script* tags are checked in the map. If the value of their *src* attribute does not end with *.js,* then 1 is added to this feature value.

To get the *presence of double documents* the number of *html, head, body* and title tags are checked. If the result is more than one in any of the mentioned tags then this feature is *true*, otherwise *false.*

For storing the HTML features, the `HTMLFeatureContainer` class is used.

JavaScript feature extraction: To calculate the JavaScript features the Abstract Syntax Tree (AST) is traversed. The AST is created by the *Esprima .NET* parser. The `Esprima.Ast.Program` has a root node the *Program*, which has a *Body* property containing the nodes in the first depth of the AST. The *EcmaScript Specification* lists the possible types of the AST nodes, and also states which nodes can contain which nodes and in which property. For example the JavaScript code `const num = 42` has the following AST in JSON **Hiba! A hivatkozási forrás nem található.**:

```
{
    "type": "Program",
    "body": [
        {
            "type": "VariableDeclaration",
            "declarations": [
                {
                    "type": "VariableDeclarator",
                    "id": {
                        "type": "Identifier",
                        "name": "answer"
                    },
                    "init": {
                        "type": "Literal",
                        "value": 42,
                        "raw": "42"
                    }
                }
            ],
            "kind": "const"
        }
    ],
    "sourceType": "script"
}
```

I wrote an anonymous function for each possible node type to traverse the tree. The functions are stored in a `Dictionary<Nodes, Action<INode, List<Nodes>, JSFeatureContainer>>` type object, where the key is the current node while

traversing the tree (`Node` is an enumeration containing every possible node name, provided by the library). The `Action` represents what should be done on the node to continue the traversing steps. The anonymous function has 3 parameters, a node which implements the `INode` interface, a list of the parent nodes in the tree, and the feature container for JavaScript features. I implemented a recursive `preOrderTraverseTree()` function, which calls the anonymous functions from the `Map`. The following `Action` has to be done on a `CallExpression` type node:

```
(node, prevNodeTypes, container) =>
{
   var n = node as CallExpression;
   container.NumOfMethodcalls++;
   //Expression
   preOrderTraverseTree(n.Callee, prevNodeTypes, container);
   foreach (var i in n.Arguments)
   {
   //ArgumentListElement
     preOrderTraverseTree
          (i as INode, prevNodeTypes, container);
   }
}
```

The given `node` is casted to `CallExpression` using the `as` keyword. The *number of methodcalls* feature is incremented, because the `CallExpression` node represents a function call. The node has two properties, which contain other nodes, and they are traversed using the function `preOrderTraverseTree()`. The code of the traversing is around 1000 line long because of the number of possible node types. The recursion ends on the following nodes: `BreakStatement`, `ContinueStatement`, `DebuggerStatement`, `EmptyStatement`, `Identifier`, `Literal`, `SpreadElement`, `Super`, `ThisExpression`, `UpdateExpression`. They do not have child nodes. The most important for the features are the `Identifiers`, and `Literals`.

The *number of method calls* feature is increased on every `CallExpression` node.

The method call based features: *number of eval calls*, *number of timer function calls, number of string modification function calls*, *number of DOM modification function calls* are extracted in an `Identifier` or a `Literal` node. In JavaScript a method can be called in three different ways: using a member expression or a call expression. For example, calling `eval`:

- `eval()`, where the `callee` of the `CallExpression` is an `Identifier`, which `Name` is "*eval*"
- `this["eval"]()`, where this refers to the global object in JavaScript. The `callee` of the `CallExpression` is a `MemberExpression`. The `MemberExpression`'s `Property` value is a `Literal` (a `StringLiteral` precisely), which `StringValue` is "*eval*".
- `this.eval()`, where this refers to the global object in JavaScript. The `callee` of the `CallExpression` is a `MemberExpression`. The `MemberExpression`'s `Property` value is an `Identifier`, which `Name` is "*eval*".

  For every function name (*setTimeout*, *createElement*, *split* etc.) these possible ways are checked in a `Literal` and an `Indentifier` node. This way an `eval` call can be hidden with a simple deobfuscation like `this["ev"+"al"]()`, this is the major drawback of static analysis.

String based features: *number of long string, number of suspicious strings, number of strings containing iframe, number of strings containing suspicious tag names, number of strings, maximum length of strings, average string length* are extracted from a Literal node, using simple string manipulation functions.

The *number of navigator properties used* feature is increased when the actual node is a `Literal` or an `Indentifier` node and the parent node is a `MemberExpression`. These are the last to possibilities mentioned in the function call based features.

The *number of long variable or function names* are increased if the `Name` of the `Identifier` node is longer than 20 characters and the previous node was `FunctionDeclaration` or `VariableDeclaration`.

The *number of Unicode characters* is calculated with the help of `Encoding.ASCII.GetByteCount()` and `Encoding.UTF8.GetByteCount()` methods.

The *average line length*, *percentage of whitespace, length of the script*, and *number of lines* features are extracted from the raw JavaScript document using simple *.NET Core Framework* functions.

Entropy based features: *average entropy of strings, max entropy of strings, entropy of script* are calculated from the found `StringLiterals`. First, the number of each character in the string is calculated. Then, the relative frequencies are calculated. Finally, the entropy is the following equation (4.1):

$$H(S) = -\sum_{c\ in\ S}^{numc} p(c) log_2(p(c)) \qquad (4.1)$$

where S is the string, c is a character in the string, $numc$ is the number of different characters in the string.

The HTML and JavaScript features are saved in two files. The first row of each file is a header row, with the name of the features (separated by tabulators). Each following row represents a feature set for one source file, also separated by tabulators. I created one additional column in the files (`IS_MALICIOUS`), which is `true` if the actual sample is malicious, otherwise `false`. This column was useful later for SPSS, and Weka.

4.2.3 Training and Validation

After the feature extraction, the set of features was available in two text files, one for the JavaScript features and one for the HTML features. Both were extended with the class type (`IS_MALICOUS`) as mentioned above.

4.2.3.1 Naïve Bayesian Classifier

I divided the set into 70-30% randomly, and used the 70% as the training set, and the 30% as the validation set. After the split, the two sets contained the number of HTML and JavaScript feature tuples given in Table 3.1.

I decided to use the SPSS Statistics program to calculate the borders of the bins, and to create the frequency tables. I imported the training set and used the Visual Binning functionality of the program to transform the values of each type of feature (columns) to their bins. I created the cutpoints of bins at `mean`, `mean+std.deviation`, `mean-std.deviation`. After the binning, I created a Custom Table for each feature, which provided the frequency tables. The frequency table of the *number of eval calls* feature can be seen in Table 4.1.

		IS_MALICIOUS			
		False		True	
		Count	Column N %	Count	Column N %
NUM_OF_EVALS (Binned)	**<= 0**	**59446**	**92.1%**	**782**	**61.5%**
	1 - 1	**2672**	**4.1%**	**291**	**22.9%**
	2+	**2411**	**3.7%**	**199**	**15.6%**

Figure 4.1. Frequency table of the *number of eval calls* feature.

After creating the frequency tables, I created the `Classification` module in C#. The structure of the module can be seen on Figure 4.3. Two classes: `JavaScriptTrainingSet` and `HTMLTrainingSet` store the frequency tables.

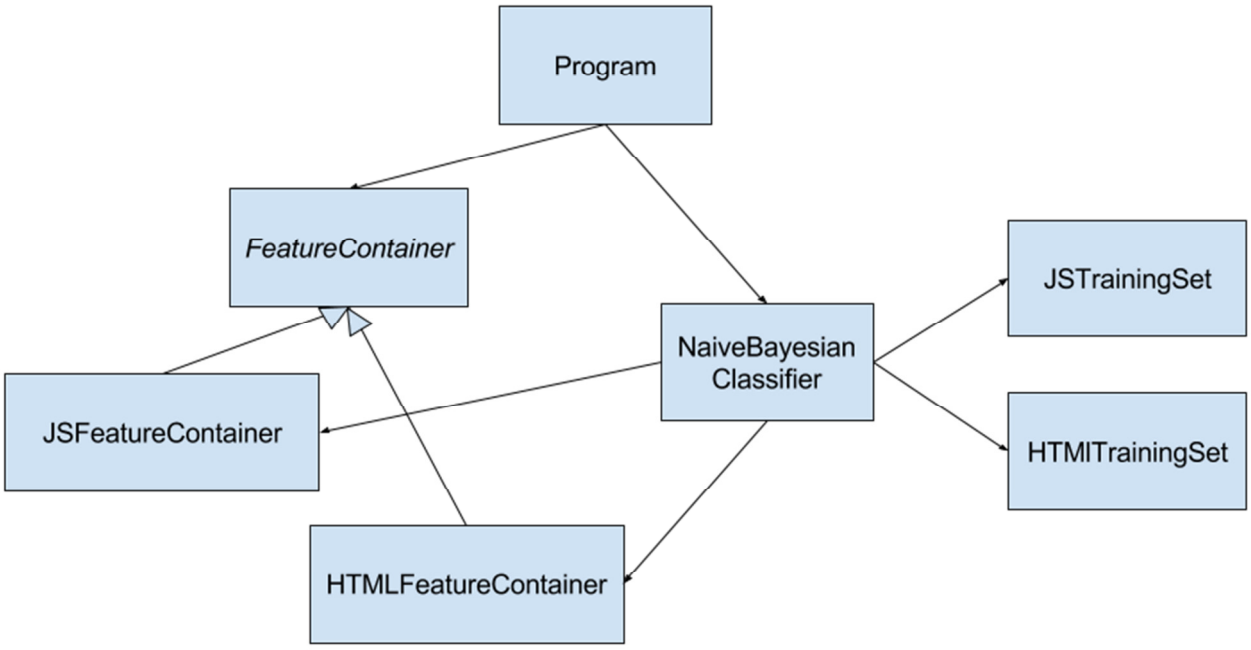

Figure 4.3. The structure of the Classifier module.

I created dictionaries for *discrete, continuous* and *logical* features:

```
Dictionary<HTMLFeatureType, Func<bool, (double, double)>>
Dictionary<HTMLFeatureType, Func<double, (double, double)>>
Dictionary<HTMLFeatureType, Func<int, (double, double)>>
```

The key of the dictionaries is the type of the feature. The possible values are available in the `HTMLFeatureType`, and the `JSFeatureType` enumerations. The value of the dictionaries is an anonymous function. The function has one input parameter; the value of the

feature (e.g.: *true, 4.3, 5*). The return value of the function is a ValueTuple containing the conditional probabilities of the sample being malicious/benign given the value of the feature.

For example the function of the *number of eval calls feature*:

```
(value) =>
{
    if (value <= 0)
          return (59446 / (double) numOfBenign, 782 / (double)
              numOfMalicious);
    else if (value <= 1)
          return (2672 / (double) numOfBenign, 291 / (double)
              numOfMalicious);
    else
          return (2411 / (double) numOfBenign, 199 / (double)
              numOfMalicious);
}
```

Where the numbers in the conditions are the borders of the bins, the numbers after the `returns` are the values in the frequency tables. The `numOfBenign` and the `numOfMalicious` variables store the number of benign and malicious samples in the training set. I imported the values in the frequency table and the borders of the bins manually.

To validate the classifier, I created the `NaiveBayesianClassifier` class, which receives an `HTMLFeatureContainer`, or a `JSFeatureContainer` and returns the probability of the sample being malicious. The `ClassifyJavaScript` and `ClassifyHTML` methods in the class query the conditional probabilities for every feature from the `TrainingSet` and multiply the values (3.4) to get the likelihoods. The last step is to normalize the likelihoods to get the actual probabilities (3.6). The return value is a probability of the sample being malicious given its feature values.

I read the validation set line by line from the program, and classified the feature tuples using the classifier. Then, I analysed the confusion matrices to get the FPR and FNR. For detailed results, the reader is referred to Section 5.

4.2.3.2 Random Forest

After seeing the good accuracy results from the Random Forest algorithm in Weka, I decided to try out and evaluate one for the filter in C#. I used a library called SharpLearning[42], which is an open source machine learning library in C#. It does not support .NET Core 1.1, only the new 2.0 release, so I upgraded the Classification module to .NET Core 2.0. The filter does not use the Random Forest classifier yet, the cause is explained in Section 5.

The library provides classes for reading feature values from CSV files, creating containers for the features, dividing the feature set to 70%-30% groups randomly, creating a Random Forest model from the feature set, predicting the classes of a test set, and creating the confusion matrix from the predicted and expected classes.

Using the provided API and example codes [49], I loaded the feature values and used the IS_MALICIOUS column as the expected label:

```
var parser = new CsvParser(() =>
        new StreamReader("js_all.csv"), ',');
var targetName = "IS_MALICIOUS";
var targets =
        parser.EnumerateRows(targetName).ToF64Vector();
var observations = parser.EnumerateRows(c =>
        c != targetName).ToF64Matrix();
```

The `observation` variable is a matrix containing the feature tuples. The `targets` variable is a list containing the expected class.

Then, I divided the feature set into 70%-30% partitions:

```
var splitter = new RandomTrainingTestIndexSplitter<double>
        (trainingPercentage: 0.7, seed: 24);

var trainingTestSplit = splitter.
        SplitSet(observations, targets);
var trainSet = trainingTestSplit.TrainingSet;
var testSet = trainingTestSplit.TestSet;
```

The `trainSet` contains 70% of the data with the observations and targets. The `testSet` contains the 30% of the data with the observations and targets.

[42] https://github.com/mdabros/SharpLearning

After this, I created the model from the `trainSet` with the default constructor parameters and classified the observations in the `testSet`:

```
var learner =
   new ClassificationRandomForestLearner(trees: 200);
var model =
   learner.Learn(trainSet.Observations, trainSet.Targets);
var predictions = model.Predict(testSet.Observations);
```

Finally, I created the confusion matrix from the predicted and expected classes and compared the results to the Naïve Bayes implementation and to the classifiers from Weka.

4.3 The Filter

This section describes the differences and changes between the modules of the filter and the modules implemented for the training. There were only small modifications in the modules; the main functionality and responsibility of the modules are already stated. The overall structure of the filter can be seen in Figure 4.4. I combined the `Crawler`, the `Feature Extractor` and the `Classifier` modules. The input parameters of the filter are the suspicious URL sources, the maximum parallel crawlers and the maximum parallel HTTP requests per crawler.

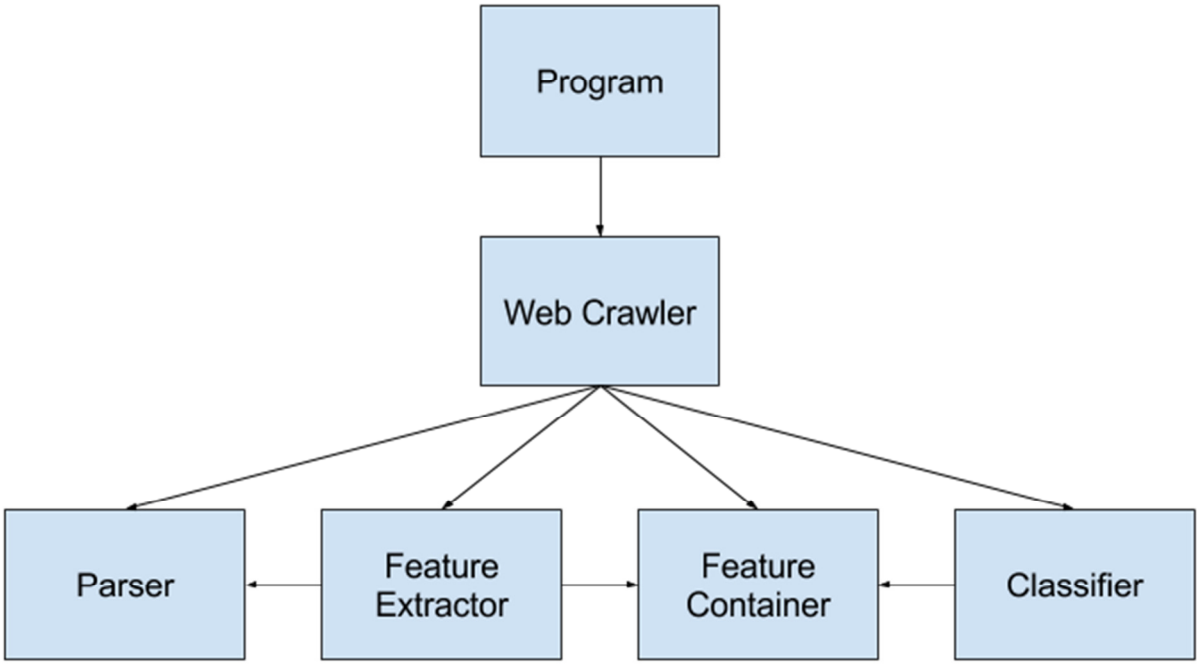

Figure 4.4. The structure of the filter.

The procedure of the classification is as follows:

1. The `Program` initializes the `Crawlers`.
2. Each `Crawler` sends HTTP requests to the specified URLs.
3. The `Crawler` receives an HTTP reply and parses the received JavaScript files too, not only the HTMLs.
4. After the parsing, the feature extraction from the parsed files is started immediately using the `Feature Extraction` module.
5. After the feature extraction the classification is started using the `Classification` module. The module returns the probability of the analysed file being malicious in case of Naïve Bayes. In case of Random Forest, it returns the predicted class.
6. The `Crawler` sends other requests or returns. Then, another `Crawler` is started.

I have done performance measurements, which were promising. For further information, the reader is referred to Section 5.

4.4 Logging

I implemented thorough logging for every module with the log4net[43] library available through NuGet. The format of a log message is the following:

```
%date [%thread] %level %logger - %message%newline
```

For example:

```
2017-07-25 12:06:21,262 [Worker 1] INFO http://ukatemi
.com - Trying to parse http://ukatemi.com/assets/js/jquery.
scrollTo-1.4.2-min.js as JavaScript on custom ThreadPool.
```

Logging is inevitable for a complex application like mine.

[43] https://logging.apache.org/log4net/

5 Results

5.1 Comparing features

I analysed the features in SPSS and Matlab and compared the distributions and frequency tables of benign and malicious feature values to find out which features can distinguish between the classes the most.

Every HTML feature was seemingly able to differentiate between the classes but the most dominant features were the following: the *number of parsing errors*, the *number of embeds,* the *number of objects* and the *presence of double document* were those, whose values in malicious files were significantly higher than in benign files. The frequency table of these 4 features can be seen in Table 5.1, where the labels next to the feature names represent the bins and their borders (except in the *presence of double document* feature, where those are the logical values).

		Benign		Malicious	
Number of parsing errors	<= 0	73441	97.5%	547	49.4%
	1 - 5	1307	**1.7%**	405	**36.6%**
	6+	613	**0.8%**	155	**14.0%**
Number of objects	<= 0	74441	98.8%	967	87.4%
	1+	920	**1.2%**	140	**12.6%**
Number of embeds	<= 0	75155	99.7%	994	89.8%
	1+	206	**0.3%**	113	**10.2%**
Presence of double document	False	69248	91.9%	726	65.6%
	True	6113	**8.1%**	381	**34.4%**

Table 5.1. The frequency table of the 4 dominant HTML features.

The *number of small elements* was the feature which was higher in benign files than in malicious files and produced the highest difference (Table 5.2).

		Benign		Malicious	
Number of small elements	<= 0	56140	74.5%	997	90.1%
	1+	19221	**25.5%**	110	**9.9%**

Table 5.2. The frequency table of the *number of small elements* feature, which was higher in benign files.

In case of JavaScript features, the most dominant ones were the method call based ones like the *number of timer function calls*, the *number of DOM modification function calls*, the *number of string modification function calls* and the *number of eval function calls*. The frequency table of these features can be seen in Table 5.3.

		Benign		Malicious	
Number of timer function calls	<= 2	49519	76.7%	999	78.5%
	3 - 9	10406	**16.1%**	73	**5.7%**
	10+	4604	**7.1%**	200	**15.7%**
Number of DOM modification function calls	<= 15	48649	75.4%	925	72.7%
	16 - 91	15032	**23.3%**	153	**12.0%**
	92+	848	**1.3%**	194	**15.3%**
Number of string modification function calls	<= 15	54490	84.4%	999	78.5%
	16 - 70	7031	**10.9%**	68	**5.3%**
	71+	3008	**4.7%**	205	**16.1%**
Number of eval function calls	<= 0	59446	92.1%	782	61.5%
	1 - 1	2672	**4.1%**	291	**22.9%**
	2+	2411	**3.7%**	199	**15.6%**

Table 5.3. The frequency table of the most dominant JavaScript features.

At first glance, the values of JavaScript features in malicious files differ more from the values in benign files than the values of HTML features. This was confirmed by the accuracy of the classifiers classifying HTML and JavaScript files.

5.2 Accuracy

The Naïve Bayesian classifier, which I implemented, returns the probability of a feature tuple being malicious. In order to use this, I define a *threshold* parameter, and if the probability is higher than the threshold, then we say that the sample is malicious, otherwise benign. By modifying the threshold parameter from 0 to 1, the confusion matrix [22] of the classifier can be created. The ROC curve is a graph, where the x axis represents the False Positive Rate, the y axis represents the False Negative Rate. The points on the curve are the FPR and FNR values using different threshold parameters. The curve can help to customize the classifier for different tasks. The ROC curve for the HTML and JavaScript features can be seen in Figure 5.1 and Figure 5.2. I labelled five points on the curve specifying the threshold value (t) and the FPR and FNR values.

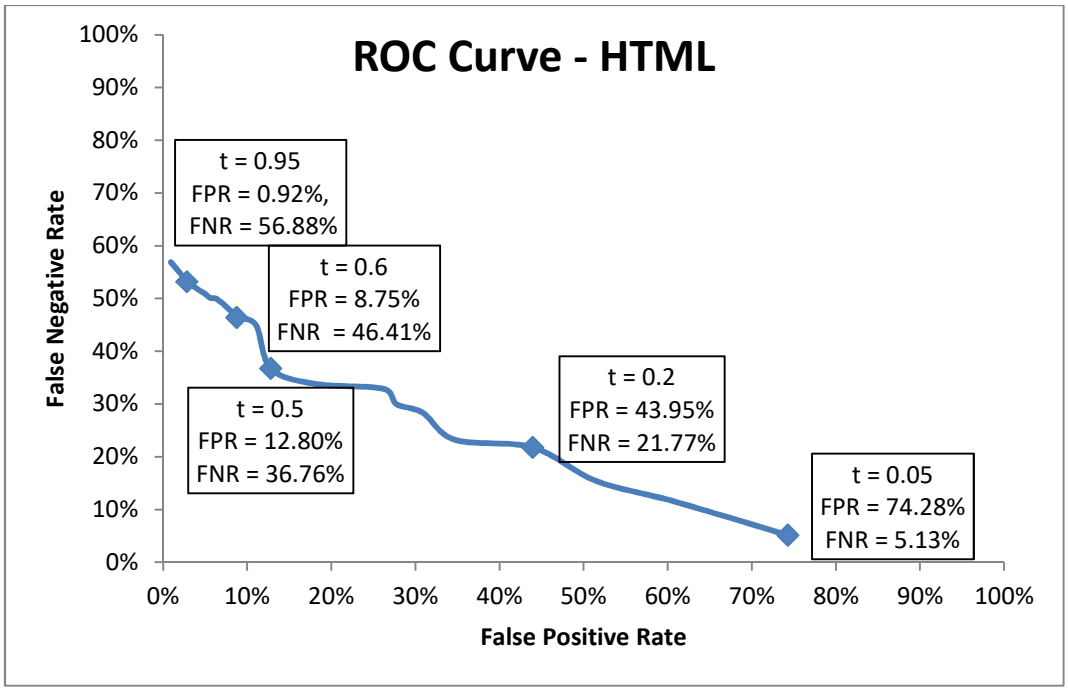

Figure 5.1. The ROC curve using the JavaScript features.

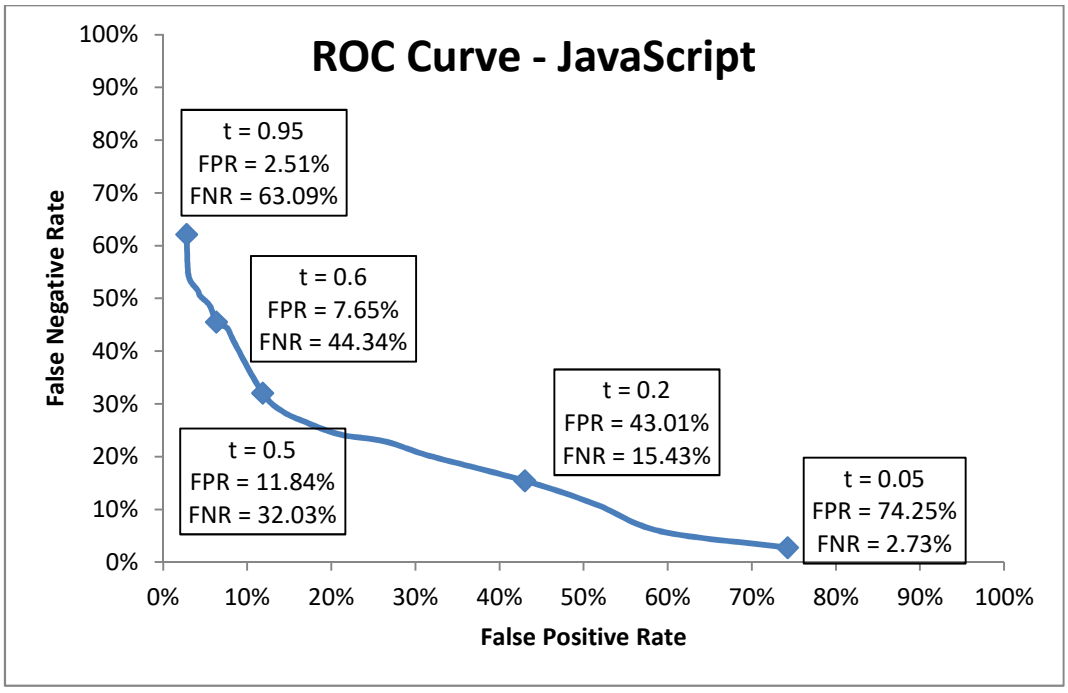

Figure 5.2. The ROC curve using the HTML features.

The curves were created from classifying the validation set (30% of all the collected samples). The ROC curves show that the classification is more accurate using JavaScript features. The FPR and FNR values for a specific threshold are 4-5% lower when classifying JavaScript. Also, as we want to lower the FNR (because we do not want to miss many malicious files and it is acceptable to classify some benign files as malicious) the FPR rate

will increase. However, when using 0.2 as a threshold value, we discard more than half of the benign files, which will increase the throughput of a dynamic analyser.

I created the confusion matrices for the 0.2 threshold and calculated the parameters listed in Section 3.3, the matrices can be seen in Table 5.4 and Table 5.5. The tables are created after classifying the validation set. It is clear that using these static features the Precision of the classifier is very low, but we did not expect higher Precision and the FNR to be low is the important criterion. Also, these tables show that the JavaScript features distinguish better between malicious and benign files than HTML features.

HTML	Labeled Malicious	Labeled Benign	
Classified malicious	**True positive** 381	**False positive** 14188	Precision, Positive Predictive Value (TP/(TP+FP)) 2.62%
Classified Benign	**False negative** 106	**True negative** 18097	Negative Predictive Value (TN/(FN+TN)) 99.42%
	Recall, Sensitivity (TP/(TP+FN)) 78.23%	Specifivity (TN/(TN+FP)) 56.05%	FPR = 43.95% FNR = 21.77%

Table 5.4. The confusion matrix using 0.2 threshold and classifying the HTML validation set.

JavaScript	Labeled Malicious	Labeled Benign	
Classified malicious	**True positive** 433	**False positive** 11909	Positive Predictive Value (TP/(TP+FP)) 3.51%
Classified Benign	**False negative** 79	**True negative** 15779	Negative Predictive Value (TN/(FN+TN)) 99.50%
	Sensitivity (TP/(TP+FN)) 84.57%	Specifivity (TN/(TN+FP)) 56.99%	FPR = 43.01% FNR = 15.43%

Table 5.5. The confusion matrix using 0.2 threshold and classifying the JavaScript validation set.

After analysing my implementation, I compared it to other classifiers. I used the Weka machine learning tool to train and validate other classifiers. I tried two methods to train and validate the classifiers. One of them was 10-fold cross validation, the other one was the 70-30% partitioning, which I used to train and validate my Naïve Bayesian classifier. The results of the two methods were almost the same, the rates only differed maximum +/- 2-3%. I present the results of the 70-30% partitioning. I tried out the following classifiers: Bayes Net, Decision Table, Logistic, J48, Random Forest, Random Tree, Naïve Bayes (3 types, using normal distribution estimation, using kernel function to estimate distribution and using discretization/binning). In the following tables I used 0.5 as the threshold of my classifier, because this way the comparison with the other Bayesian classifiers is more precise.

Figure 5.3 shows the False Negative Rate of each classifier in case of HTML samples, my classifier is the 1st from the right. The figure shows that my classifier was the 4th best comparing to others, the Random Tree, the Random Forest and the J48 classifiers produced the lowest FNRs. By setting the threshold to 0.2 my classifier produces 21.77% FNR. The normal distribution based, and the kernel function based Naïve Bayesian classifiers perform poor, so it was a good choice not to estimate the distributions of the feature values. In case of Precision, the Random Forest achieved the maximum: 97.63%. Combining Precision and Recall (1-FNR) to get F-measure, the Random Forest was the best, reaching 84.40%. My classifier performed poor in Precision as mentioned before, but we focused on FNR.

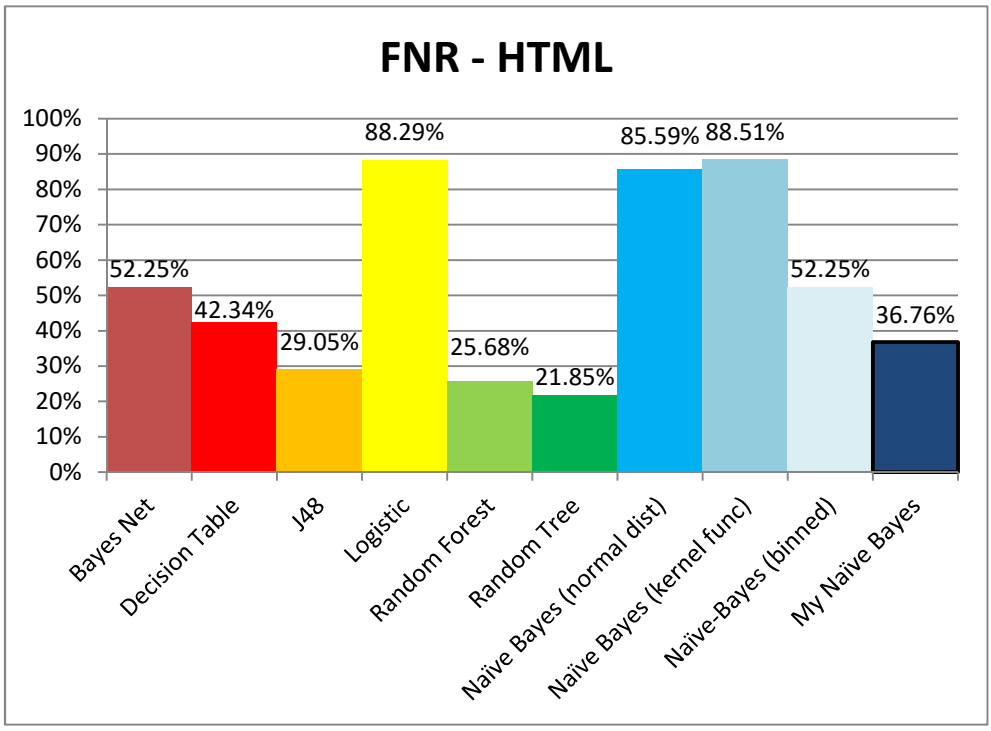

Figure 5.3. The False Negative Rate of the classifiers used using HTML features. Using t=0.5 for my Naïve Bayes.

Figure 5.4 shows the FNR of the classifiers using JavaScript features, my classifier is the 1st from the right. The FNR rates were 4.8% lower using JavaScript features than using HTML features. The results were the same as using HTML features. The other Bayesian classifiers performed poor, the Random Forest, Random Tree and J48 classifiers were the best, and mine were the 4th. In case of JavaScript features the Random Forest produced the highest Precision (99.80%) and the highest F-measure (90.33%).

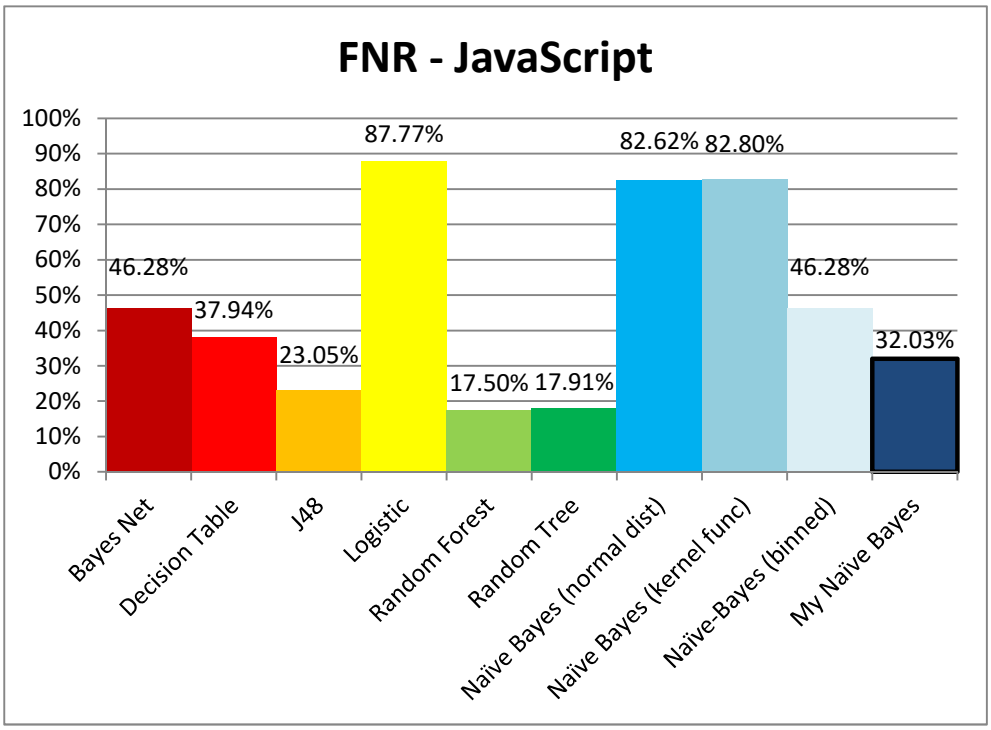

Figure 5.4. The False Negative Rate of the classifiers using JavaScript features. Using t=0.5 for my Naïve Bayes.

We can draw two main conclusions from the accuracy results. The first one is that my implementation has a good FNR compared to others and modifying the threshold it can produce lower FNR than the other classifiers, this is good for our task. However, taking Precision and F-measure into account my classifier was poor, using a Random Tree, or Random Forest classifier is a possible way for the future. The second one is that using the selected features the Random Forest, the Random Tree and the J48 performed very well both in Precision and Recall.

After analysing the result, I trained and evaluated a Random Forest algorithm in C#. The Naïve Bayesian Classifier has configurable FNR, but to reach 10% FNR the FPR rate will increase to 50% percent, which means that we can only halve the benign pages passed to the dynamic analyser. Assuming that malicious pages are rare, this means that we could only

double the throughput of the dynamic analyser. Lower than 1% FPR would be the best, which would mean that we could increase the throughput around 100 times.

The Random Forest provided promising results. The FNR, FPR, Precision, Recall and F-measure rates compared to the best 4 classifiers in Weka and compared to my Naïve Bayes implementation (with 0.2 threshold) can be seen in Table 5.6 and Table 5.7 .

HTML Features	FPR	FNR	Precision	Recall	F-measure
Decision Table	0.03%	42.34%	96.60%	57.66%	72.21%
J48	0.09%	29.05%	91.57%	70.95%	79.95%
Random Forest	0.02%	25.68%	97.63%	74.32%	84.40%
Random Tree	0.32%	21.85%	76.94%	78.15%	77.54%
My Naïve Bayes	**43.95%**	**21.77%**	**2.62%**	**78.23%**	**5.06%**
My Random Forest	**0.02%**	**29.32%**	**98.48%**	**70.68%**	**82.29%**

Table 5.6. The FPR, FNR, Precision, Recall and F-measure values of the best classifiers from Weka, and the Naïve Bayes (with t=0.2) and Random Forest classifier used in the filter using HTML features.

JavaScript Features	FPR	FNR	Precision	Recall	F-measure
Decision Table	0.00%	37.94%	99.72%	62.06%	76.50%
J48	0.18%	23.05%	89.67%	76.95%	82.82%
Random Forest	0.00%	17.50%	99.80%	82.50%	90.33%
Random Tree	0.48%	17.91%	77.82%	82.09%	79.90%
My Naïve Bayes	**43.01%**	**15.43%**	**3.51%**	**84.57%**	**6.74%**
My Random Forest	**0.01%**	**23.17%**	**99.49%**	**76.83%**	**86.70%**

Table 5.7. The FPR, FNR, Precision, Recall and F-measure values of the best classifiers from Weka, and the Naïve Bayes (with t=0.2) and Random Forest classifier used in the filter using JavaScript features.

The Random Forest implementation in C# reached the best classifiers in accuracy, with around 0.01% FPR and 25% FNR.

The current configuration of the filter uses my Naïve Bayes implementation with 0.2 threshold, because the most important aim is not to miss many malicious pages, but more than half of the benign pages are discarded. The Random Forest algorithm was evaluated with the provided default parameters (e.g.: 100 trees, 2000 maximum tree depth, 0.00001 minimum information gain before a split is made). We need further optimization to create the best model and to lower the FNR before using the Random Forest as the classifier of the filter.

5.3 Performance

Besides the low False Positive Rate, the other requirement for the filter was to be able to analyse more than 10.000 or even more than 100.000 URLs per day. I started the filter to analyse around 100.000 URLs. The test performance run was carried out on a virtual machine with 2 CPU cores. I used my seed URL list containing around 400 URLs to run a scan, and I set the filter to analyse at least 300 URLs per seed. I started 10 crawlers simultaneously; each of them sent 20 HTTP requests parallelly. It took 188 minutes to classify 118889 files. This means around 95ms per URL, and **909500 URL per day**. During the analysis, each crawler spent around 50-60% of their time sending, waiting for and processing requests, 30-40% of the time with feature extraction and 10% of the time parsing the files. The time consumed by classification is negligible; it took only a few milliseconds per crawler (either with Naïve Bayes or Random Forest). These results make it possible to use the filter to classify a million URLs per day and pass the potentially malicious ones to a dynamic analyser.

6 Future Work

I believe that much remains to be done in my work. In this section I am going to list the possible future improvements I have thought of.

6.1 Improving Throughput

The performance results clearly showed that the most time consuming tasks are: waiting for HTTP replies, extracting features and parsing the files. Increasing and optimizing the requests sent simultaneously can increase the throughput. Also, the custom `ThreadPool` I used for feature extraction can be implemented in the filter too.

6.2 Filtering the Training Set

As I have mentioned before, I did not manually analyse the samples of the benign and the malicious training and validation set.

Benign samples: With around 100.000 HTML and JavaScript benign samples, the manual analysis is infeasible, but it is not needed. It is a good assumption that crawling the most popular web sites will not result malicious samples. Also, to remove duplicate files, matching the URLs, and the SHA-256 hashes is also considered enough.

Malicious samples: The malicious set is more problematic. I only identified the samples with their SHA-256 hash value, so it is possible that there are similar samples. Hashing files which only differ in a whitespace at the end, or a string containing a timestamp will result in completely different hash values. Also, since I collected samples only in 2 months' time (2017 August - September) it is possible that the result set contains similar attack types and exploits, so the classifier only learned those types. By manually analysing the malicious set and collecting other samples, the precision of the classifier could be increased.

6.3 Modifying Features

Selecting those features, which describes best the differences between malicious and benign files is important. In the future there are three possible ways to make improvements with the features:

- Removing those features, which values are very similar in malicious and benign files, based in their distributions and other statistical properties.
- Selecting and implementing more sophisticated features like: *number of event attachments, shellcode presence probability* or *presences of deobfuscation routines* [16].
- Besides analysing the JavaScript and the HTML source checking the URL or the *whois*[44] information of the host of the examined web page can also be interesting [20].

6.4 Training Other Classifier

Naïve Bayesian Classifier was a good choice at first, for creating a prototype, because it is easy to understand, train and implement and it is also one of the fastest classifiers. The False Negative Rate was good compared to other classifiers. However, the precision of a Naïve Bayesian Classifier is quite poor compared to other, more sophisticated classifiers like J48, Random Forest and Random Tree. This was clearly seen in the results of the Weka tool, and from the related works. With other classification methods the filter could produce lower false positives, lower false negatives and better precision. Optimizing the Random Forest algorithm and substituting the Naïve Bayesian Classifier with it is a possible way.

[44] https://en.wikipedia.org/wiki/WHOIS

7 Conculsions

As the web-based malware distribution spreads, there is a need for fast and precise malware detection. The state of the art malware detection technique is to use a dynamic analyser, which loads the page and executes the JavaScript code. This is usually done on isolated virtual machines. The problem with dynamic analysers is that they are slow and they cannot check hundred thousand URLs per day. Our aim was to create a static analyser, which classifies a page only using its lexical and syntactical parameters, without any execution. This analyser can be used as a filter for a dynamic analyser, or a filter used in web browsers.

Our approach is based on previous research results. We collected benign files using a web crawler, which we implemented. We gathered malicious files, using VirusTotal's notification service and Ukatemi's malware database. We defined 16 HTML and 22 JavaScript features, and extracted them from the collected samples, which provided the training set. After the extraction, we implemented a Naïve Bayesian Classifier, trained it and validated it.

The two main requirements from the filter were high throughput, and low False Negative Rate. The filter contains three modules. The first one is a web crawler, which downloads and parses the given URLs. The second one is responsible for extracting features from the parsed JavaScript and HTML files. The third module is the Naïve Bayesian classifier used for classifying the downloaded files using the extracted features. All three modules were implemented by me.

The results were promising. The filter was able to analyse around a million URLs per day. We also tried and compared our classifier to others using the Weka machine learning tool. Our classifier was able to achieve around 15-30% False Negative Rate, which was similar to the previous results. JavaScript features produced better results than HTML features. There were three classifiers, Random Tree, Random Forest and J48, which outperformed our implementation and produced more than 80% Precision and Recall. This means that using static features it is possible to classify HTML and JavaScript files accurately. Seeing the results, I also integrated a Random Tree classifier with the filter, which achieved 99% Precision and 20-25% FNR, but we need further optimization to substitute the Naïve Bayesian Classifier.

There are multiple ways to develop and optimize the filter. Using another classifier, rather than Naïve Bayes is a possible way. Also, modifying the feature set can help to lower the False Positive Rate and to increase the Precision.

Acknowledgements

First of all, I would like to thank my supervisor Dr. Levente Buttyan. He convinced me to work on this paper and submit it to the Scientific Students' Conference. Not only was he always available for discussion, but he also proposed improvements, which helped to increase the quality of this document.

I also thank Dr. Boldizsar Bencsath and Oliver Daniel Guba from Ukatemi Technologies and BME for helping me out when I encountered technical difficulties.

And last but not least I am grateful to my family. They supported me and respected my decision of spending long hours working on this paper.

References

All provided pages were visited between August and October 2017.

[1] Microsoft Secure Blog, *What you should know about drive-by-download attacks,* https://blogs.microsoft.com/microsoftsecure/2011/12/08/what-you-should-know-about-drive-by-download-attacks-part-1/, 2011.

[2] SecureMac, *Five Malware Distribution Methods and How to Protect Against Them,* https://www.securemac.com/checklist/five-malware-distribution-methods-protect, 2017.

[3] University of Delaware, *How is malware distributed?* http://sites.udel.edu/infosecnews/2015/05/18/how-is-malware-distributed/, 2015.

[4] Zero Hacks, *What is phishing?* | *How to create phishing page* | *Facebook example*, https://www.7xter.com/2016/08/phishing.html, 2017.

[5] Kaspersky Lab, *What is Spear Phishing? – Definition*, https://usa.kaspersky.com/resource-center/definitions/spear-phishing.

[6] Cyber Security Community, *Watering Hole Attack - A Sophisticated Alternate to Spear Phishing Attack*, https://securitycommunity.tcs.com/infosecsoapbox/articles/2017/02/06/watering-hole-attack-sophisticated-alternate-spear-phishing-attack, 2017.

[7] CVE Details - The ultimate security vulnerability datasource, http://www.cvedetails.com/product/9900/Microsoft-Internet-Explorer.html?vendor_id=26.

[8] Weka 3: Data Mining Software in Java, https://www.cs.waikato.ac.nz/ml/weka/

[9] P. Ratanaworabhan, B. Livshits and B. Zorn, *Nozzle: A Defense Against Heap-spraying Code Injection Attacks*, https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2008-176.pdf, 2008.

[10] Microsoft Research, *Detection of JavaScript-based Malware,* https://www.microsoft.com/en-us/research/project/detection-of-javascript-based-malware/, 2008.

[11] Microsoft Research, *Detours: Binary Interception of Win32 Functions* https://www.microsoft.com/en-us/research/project/detours/, 1999.

[12] A. Ikinci, T. Holz and F. Freiling, *Monkey-Spider: Detecting Malicious Websites with Low-Interaction Honeyclients*, http://www.dihe.de/docs/docs/monkey-spider-Sicherheit08.pdf, 2008.

[13] Quttera Crawler, *CLI-based URL scanner for Windows*, http://quttera.blogspot.hu/2013/03/cli-based-url-scanner-for-windows.html, 2013.

[14] M. Cova, C. Kruegel and G. Vigna, *Detection and Analysis of Drive-by-Download Attack and Malicious JavaScript Code*, https://seclab.cs.ucsb.edu/media/uploads/papers/www10_jsand.pdf, 2011.

[15] B. Feinstein and D. Peck, *Caffeine Monkey: Automated Collection , Detection and Analysis of Malicious JavaScript*, https://www.blackhat.com/presentations/bh-usa-07/Feinstein _and_Peck/Whitepaper/bh-usa-07-feinstein_and_peck-WP.pdf, 2007.

[16] C. Curtsinger, B. Livshits, B. Zorn and C. Seifert, *Zoozle: Low-overhead Mostly Static JavaScript Malware Detection*, https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-1-11-11.pdf, 2010.

[17] Microsoft Research, *Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities*, https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2005-72.pdf, 2005.

[18] J. Nazario, *PhoneyC: A Virtual Client Honeypot*, https://www.usenix.org/legacy/event/leet09/tech/full_papers/nazario/nazario.pdf, 2009.

[19] P. Likarish, E. Jung and I. Jo, *Obfuscated Malicious Javascript Detection using Classification Techniques*, https://www.researchgate.net/profile/Peter_Likarish/publication/224110475_Obfuscated_malicious_JavaScript_detection_using_classification_techniques/links/0c96052311027562ed000000.pdf, 2009.

[20] D. Canali, M. Cova, G. Vigna and C. Kruegel: *Prophiler: A fast filter for the large-scale detection of malicious web pages,* https://hal.archives-ouvertes.fr/file/index/docid/727271/filename/www2011_Prophiler_a_fast_filter_for_the_large_scale_detection_of_malicious_web_pages.pdf, 2011.

[21] D. Klein: Introduction to Classification: Likelihoods, Margins, Features, and Kernels, 2007, https://people.eecs.berkeley.edu/~klein/papers/classification-tutorial-naacl2007.pdf, 2007.

[22] D. M W Powers, *Evaluation: From Precision, Recall and F-Factorto ROC, Informedness, Markedness & Correlation*, http://www.flinders.edu.au/science_engineering/fms/School-CSEM/publications/tech_reps-research_artfcts/TRRA_2007.pdf, 2007.

[23] IANA, MIME-types, https://www.iana.org/assignments/media-types/media-types.xhtml.

[24] How2Lab, Iframe Hacking, http://www.how2lab.com/internet/security/iframe-hacking.php.

[25] OWASP, XSS Filter Evasion Cheat-Sheet, https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet.

[26] W3C Consortium, HTML 4 Document Type Definition, https://www.w3.org/TR/html401/sgml/dtd.html.

[27] W3Schools, HTML <meta> http-equiv Attribute, https://www.w3schools.com/TAGs/att_meta_http_equiv.asp.

[28] J. Schneider, *Cross Validation*, https://www.cs.cmu.edu/~schneide/tut5/node42.html.

[29] Sucuri Blog, *Analysing a Malicious iFrame – Following the Eval Trail*, https://blog.sucuri.net/2014/05/analyzing-a-malicious-iframe-following-the-eval-trail.html, 2014.

[30] Kahu Security, *Javascript Deobfuscation Tools Redux*, http://www.kahusecurity.com/2014/javascript-deobfuscation-tools-redux/, 2014.

[31] ECMA International, *ECMAScript® 2017 Language Specification (ECMA-262, 8th edition, June 2017)*, https://www.ecma-international.org/ecma-262/8.0/index.html#sec-ecmascript-language-expressions.

[32] Panda Security, *Deobfuscating malicious code layer by layer*, https://www.pandasecurity.com/mediacenter/malware/deobfuscating-malicious-code-layer-by-layer/, 2011.

[33] NetScope Inc., *Manually Deobfuscating Strings Obfuscated in Malicious JavaScript Code*, https://www.netskope.com/blog/manually-deobfuscating-strings-obfuscated-malicious-javascript-code/, 2016.

[34] W. Alcorn, C. Frichot, M. Orru, *The Browser Hacker's Handbook*, page 111, https://books.google.hu/books?id=lXr0AgAAQBAJ&pg=PA110&source=gbs_toc_r&cad=3#v=onepage&q&f=false, 2014.

[35] R. R. Bouckaert, Bayesian networks in Weka, Technical Report 14/2004, https://www.cs.waikato.ac.nz/~remco/weka.bn.pdf, 2004.

[36] G. H. John, P. Langley, Estimating Continuous Distributions in Bayesian Classifiers, http://web.cs.iastate.edu/~honavar/bayes-continuous.pdf, 1995.

[37] S. le Cessie, J. C. van Houwelingen, Ridge Estimators in Logistic Regression, http://www.inf.unibz.it/dis/teaching/DWDM/project2010/LogisticRegression.pdf, 1992.

[38] R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, CA. 1993.

[39] L. Breiman, *Random Forests*, https://www.stat.berkeley.edu/~breiman/randomforest2001.pdf, 2001.

[40] K. Ming Leung, *Naive Bayesian Classifier*, http://cis.poly.edu/~mleung/FRE7851/f07/naiveBayesianClassifier.pdf, 2007.

[41] Jun-Kai Yi, Yang-Ping Zhang, Xiang-Hui Zhao, *Spam Recognition Based on Bayesian Classification*, http://dpi-proceedings.com/index.php/dtcse/article/viewFile/5707/5325, 2016.

[42] L. Breiman, A. Cutler, *Random Forests*, https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm

[43] SharpLearning, *Hyperparameter Tuning*, https://github.com/mdabros/SharpLearning/wiki/Hyperparameter-Tuning

[44] Wikipedia, *Random forest*, https://en.wikipedia.org/wiki/Random_forest

[45] R. Lander, *Announcing .NET Core 1.0*, https://blogs.msdn.microsoft.com/dotnet/2016/06/27/announcing-net-core-1-0/, 2016

[46] MSDN Library, HttpClient Class, https://msdn.microsoft.com/en-us/library/system.net.http.httpclient(v=vs.118).aspx.

[47] Microsoft Docs, *Asynchronous Programming*, https://docs.microsoft.com/en-us/dotnet/csharp/async.

[48] StackOverFlow, *Code for a simple thread pool in C#,* https://stackoverflow.com/questions/435668/code-for-a-simple-thread-pool-in-c-sharp/436552#436552, 2013

[49] SharpLearning, *Introduction to SharpLearning*, https://github.com/mdabros/SharpLearning/wiki/introduction-to-SharpLearning

Appendix

The Yara rule used for gathering malicious HTML and JavaScript files from VirusTotal:

```
rule js:js {

strings:

$php="<?"
$mz="MZ"
$pk="PK"
$c1="function"
$c2="this."
$c3="()"
$c4="var"

condition:
(not $mz at 0) and (not $pk at 0) and (3 of ($c*)) and not
$php

}
```