Imperial College London

Department of Computing

Detecting Malicious Web Content with Machine Learning

Author:
Antoine Vianey-Liaud

Supervisor:
Dr. Sergio Maffeis

Submitted in partial fulfillment of the requirements for the MSc degree in Computing (Secure Software Systems) of Imperial College London

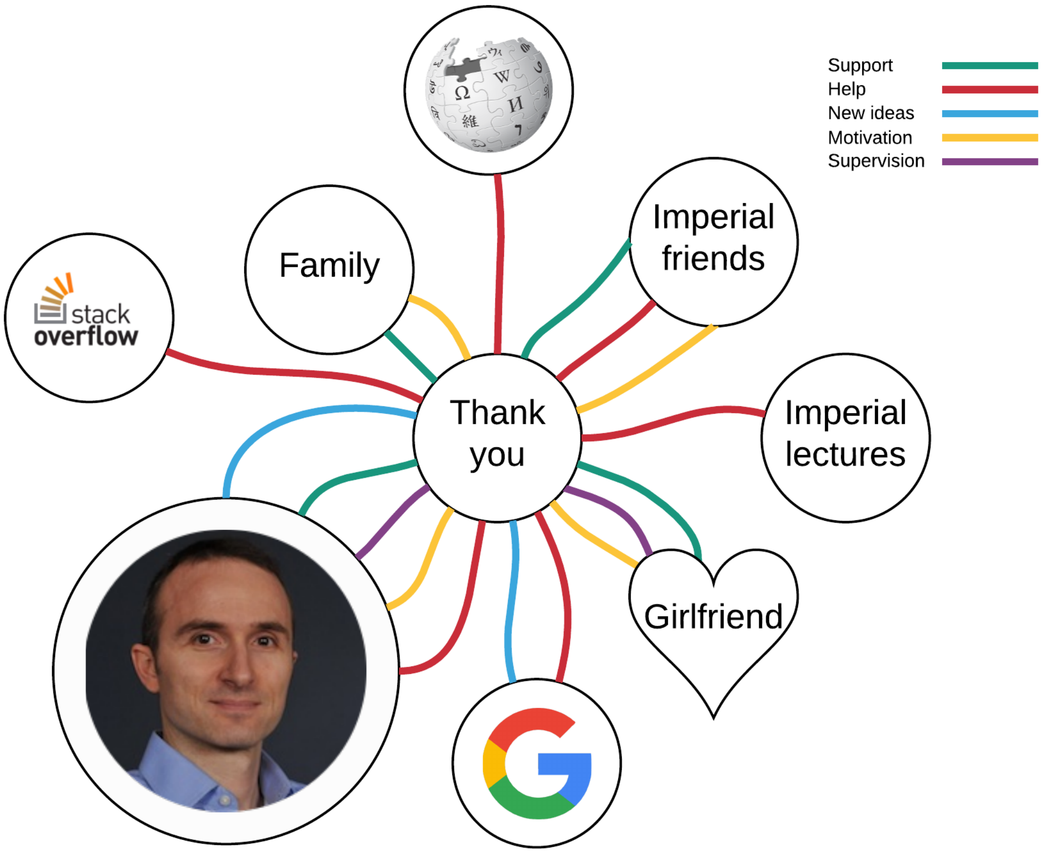
September 9, 2016

Abstract

As more and more people are connected through the web, attackers make use of more and more sophisticated methods to steal, tamper or delete critical data. On the other side, defenders must be one step ahead and protect information the best they can. There is therefore an endless arm race between attackers and defenders: static detectors like antiviruses are defeated by obfuscation, obfuscation is detected through dynamic analysis, but dynamic analysis can be evaded by malicious software thanks to fingerprinting, and this evasion is spotted by new anti-evasive approaches. This project introduces state-of-the-art methods formalised in the literature to detect malicious websites through the internet thanks to machine learning classification.

After selecting the best practices in the literature, we propose an implementation of a new detector of malicious webpages. We collect 440 bengin URLs from the 500 most popular websites in the United Kingdom and 257 malicious URLs from two blacklists hosted on the web. 23 static features are extracted from the collected data and feed four machine learning classifiers: Support Vector Machine (SVM), Decision Tree, Random Forest and AdaBoost, from which we perform a 5-cross validation and get a bunch of validation metrics. Random Forest gives the best results with 5% false negative (websites flagged as benign by the detector but which are in fact malicious), 12% false positive (websites flagged as malicious by the detector but which are in fact benign) and precision and recall of respectively 83% and 92%. The HTML letters and words count and the number of words per line in Javascript files hosted in a page are the features that have the most impact on the classification. This is due to the fact that the structure of the pages collected in the benign dataset (popular websites containing a large amount of data) are very different from those collected in the malicious dataset (less popular website containing less content). This issue explains the high rate of false positives since benign websites with a few amount of content are more likely to be considered as malicious. We made an online version of the detector: it runs on a web server, accessible by a specific URL and a Firefox extension. Finally, we draw a tree of malicious and benign URLs accessible from a given base page and at a certain "click depth": one has therefore an indication on the probability of visiting a malicious URL when clicking on a link derived from the base webpage.

Acknowledgements



Contents

List of Figures

List of Tables

Chapter 1

Introduction

In a world where almost half of the population uses the internet [1], the security of the growing amount of data exposed is crucial. Attackers take advantage of the web to steal, tamper or delete sensitive information in order to either make money, achieve political aims or even to show their talent to others.

To achieve their goal, attackers often use malicious software, also known as malware. Symantec indeed reported that 317 million new pieces of malware have been created in 2014 and 1 out of 1126 websites is somehow infected with malware [2]. Attackers use more and more advanced techniques to spread malware across the web, that is why defenders must not rest on their laurels and keep innovating to mitigate such threats.

One of the most early malware attacks was the so-called *ILOVEYOU* worm (also known as *Loveletter* and *The Love Bug*) [3]. Created by Filipino hackers, it used a fake love letter attachment (`Love-Letter-for-you.txt.vbs`) to hide a malicious VBS script which would infect image files, email itself repeatingly and destroy storage partitions. As it only needed a starting and an ending point, these worms kept self-replicating using Outlook with no need of extra software. It infected a total of 10% of the network which represents around 5 billion dollars.

This project is based on the survey “Machine Learning Techniques for Malicious Web Content Detection”, written as part of my Independent Study Option. The survey was supervised by Dr. Sergio Maffeis and recapitulates several learning techniques used by defenders to detect and mitigate different sorts of attacks and web-based malware. The aim here is to take advantage of the knowledge collected in the literature by selecting the best practices and elaborating of a new detector of malicious web content.

We describe first different approaches found in the literature and select the best practices. Then we depict the methodology we use to create our new detector: from data collection, to feature extraction, classification and prediction. After that, we show the implementation of the approach and explain how our online detector makes use of a web server and a browser extension to predict whether a web page is malicious or not. Finally, we analyse the performance of our detector in terms of running time and validation metrics, and compare them to existing techniques.

All the code and examples explained in the report can be found in the github repository: `https://github.com/antoinevl/MSc-project`.

Chapter 2

Background

2.1 Malicious content

2.1.1 Malware

Kapersky [4] classifies the different types of malware given the "threat" it poses. Figure 2.1 recapitulates this classification: whereas exploits are less likely to trigger a big threat to the system, rootkits, trojans and backdoors are more dangerous, but still less harmful than viruses and worms.

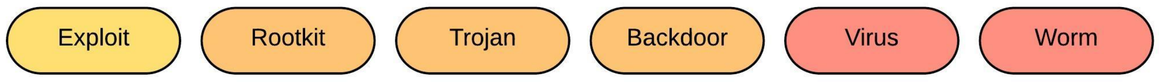


Figure 2.1: Some types of malware. Yellow poses the least threat, red pose greater threat [4]

One more global class of malware widely seen in the wild is drive-by-download. To Cova et al. [5], drive-by-download attacks are that successful because of:

- the abundance of existing vulnerabilities targeting browsers: 74 CVE (Common Vulnerabilities and Exposures) [6] entries affecting browsers have been reported only in 2015.
- the rich documentation to take advantage of these vulnerabilities: forums, videos, IRC channels etc.
- the use of sophisticated methods fingerprinting the victim's browser and then avoiding classical detectors: a popular example is Panopticlick [7], an online fingerprinting tool that detects a large range of information on users' browsers.

2.1.2 Attacks

Attackers have several ways to spread malware. One of the most prolific attacks is Cross-Site Scripting (XSS) [8] [9].

Cross-Site Scripting (XSS) consists in injecting malicious code on client side through legitimate web pages. There are three types of XSS attacks: Stored, Reflective and DOM-based. The first one designates a script injected and stored on the server. The second one refers to scripts executed only on the client browser but not affecting the server. And the third one modifies the DOM (Document Object Model) environment of a web page. Figures 2.2 and 2.3 are examples of respectively reflected and stored XSS.

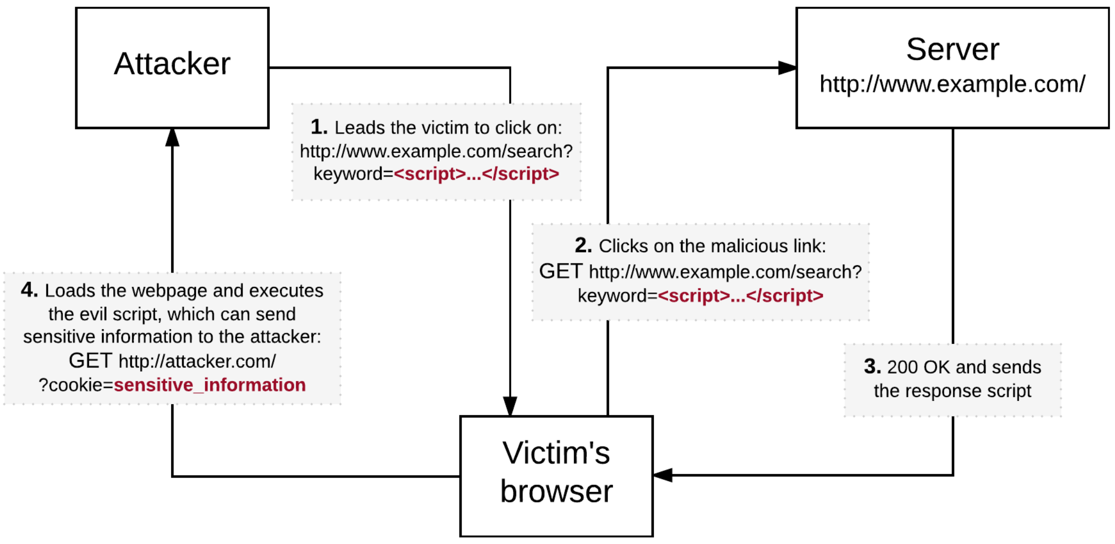

Attacker
Server
http://www.example.com/
1. Leads the victim to click on: http://www.example.com/search?keyword=<script>...</script>
2. Clicks on the malicious link: GET http://www.example.com/search?keyword=<script>...</script>
3. 200 OK and sends the response script
4. Loads the webpage and executes the evil script, which can send sensitive information to the attacker: GET http://attacker.com/?cookie=sensitive_information
Victim's browser

Figure 2.2: Example of reflected XSS

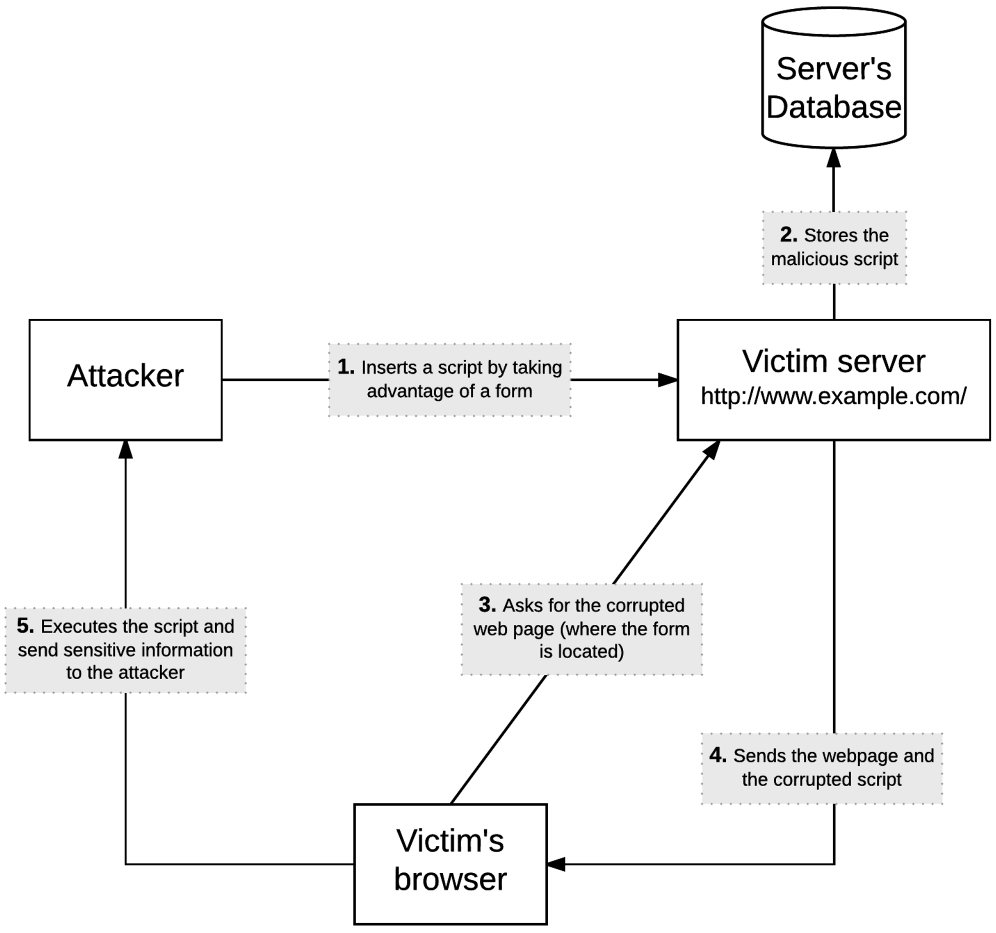

Server's Database
2. Stores the malicious script
Attacker
1. Inserts a script by taking advantage of a form
Victim server
http://www.example.com/
3. Asks for the corrupted web page (where the form is located)
5. Executes the script and send sensitive information to the attacker
4. Sends the webpage and the corrupted script
Victim's browser

Figure 2.3: Example of stored XSS

2.2 An endless arm race

The internet is an inexhaustible source of sensitive information obviously exposed to attacks. On one hand attackers keep innovating to always have a head start on defenders, and on the other hand, defenders find more and more elaborated mechanisms to detect malicious behaviours. This is an actual arm race between attackers and defenders that is represented by Figure 2.4.

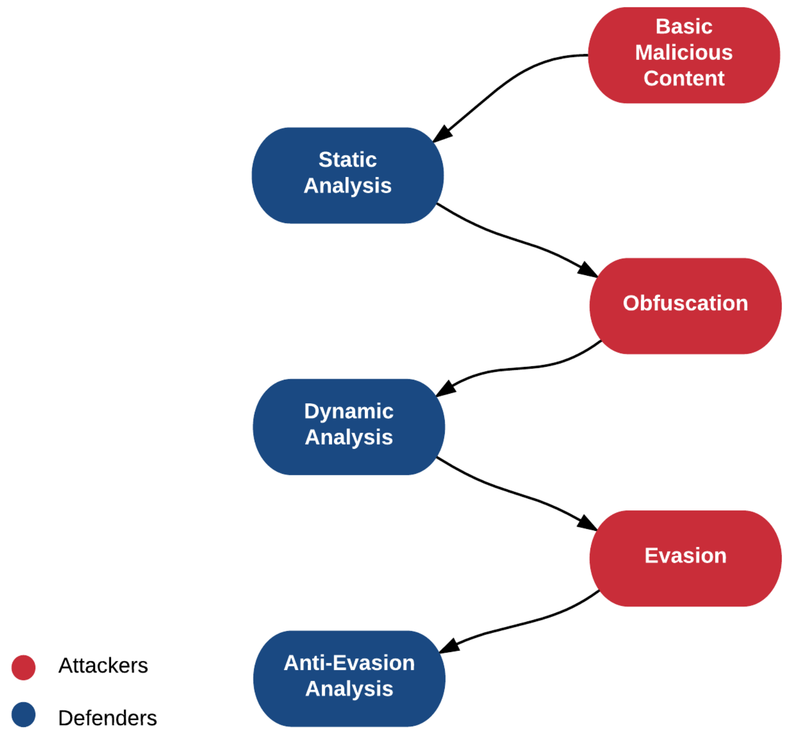


Figure 2.4: Representation of the armrace between attackers and defenders

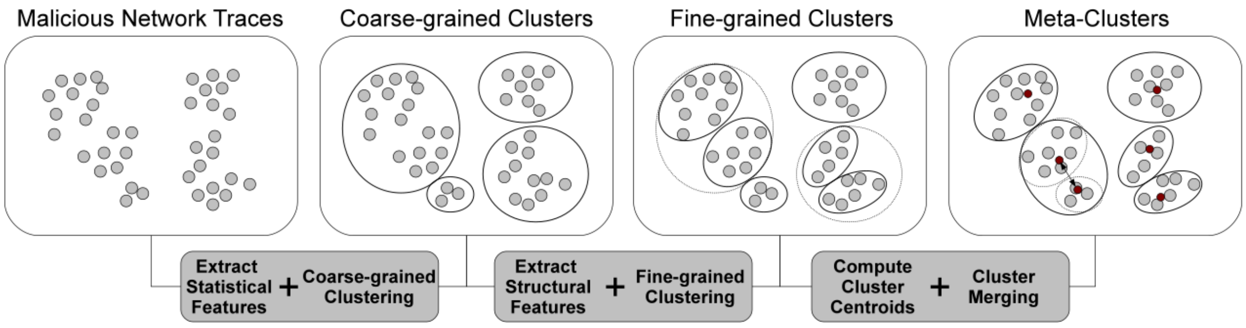
2.2.1 Static analysis

A first defense mechanism against web-based malware is static analysis such as rule-based or regular-expression-based methods that look for keywords and specific structures in malicious code (features like URIs' length, number of specific tags used etc.).

Jovanovic et al. [10] implemented Pixy, a static analysis tool that detects taint-style vulnerabilities, and more particularly XSS vulnerabilities, in PHP code. Let us recall that *tainted* data is data originated from a potentially malicous user and able to trigger malicous behaviours and target *sensitive sinks* of the program. Briefly, Pixy a data flow analysis that statically computes a number of information at each point of the code. Here, a taint analysis is made which consists in finding where tainted values can be entered in the program and what parts of the program they can affect.

Another example is the use of malware signatures generated and stored into databases owned by antiviruses.

Static analysis is still an active research domain nowadays with a simple problematic: how to generate signatures precise enough to detect all pieces of malicious software and general enough to be able to detect a slightly modified malicious script? Indeed, attackers often use polymorphic methods to modify the structure of their malware and then evade signature-based mechanisms. Perdisci et al. [11] worked in 2010 on a new way to cluster HTTP-based malware and generate network-level signatures for them. Studying network-level approaches for malware detection is motivated by the ease of intercepting packets thanks to existing network monitoring tools. Once intercepted, a malicious traffic of packets goes through several levels of clustering described on Figure 2.5.



Malicious Network Traces
Coarse-grained Clusters
Fine-grained Clusters
Meta-Clusters
Extract Statistical Features + Coarse-grained Clustering
Extract Structural Features + Fine-grained Clustering
Compute Cluster Centroids + Cluster Merging

Figure 2.5: Static clustering [11]

The first step is a coarse-grained clustering made from the analysis of simple features such as the total number of HTTP requests generated, the number of GET and POST requests or the average length of URLs.
Then, a fine-grained clustering is done by computing the distance d_r between HTTP queries. In particular: $d_r(q_1, q_2) = w_m d_m(q_1, q_2) + w_p d_p(q_1, q_2) + w_n d_n(q_1, q_2) + w_v d_v(q_1, q_2)$ where q_1 and q_2 are two HTTP queries, d_m, d_p, d_n, d_v are distance functions corresponding to respectively the request method (GET, POST etc.), path and page name (not including parameters), parameter names, parameter values. An example is shown in Figure 2.6.

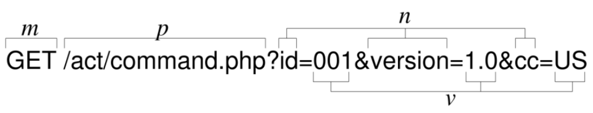


Figure 2.6: Fine-grained clustering [11]. m is the request method, p is the page name, n is the set of parameter names, v is the set of parameters values

Signatures are generated for each item thanks to the *Token-Subsequences* algorithm [12]. As shown on Figure 2.7, a signature is a list of tokens `ti` written as a regular expression of the kind: `t1.*t2.*...*tn`.

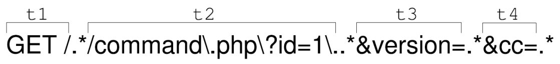


Figure 2.7: Example of signature [11]

Finally, clusters centroids are computed from their sets of signatures and some of them are merged in order to make sure that clusters are not too small (according to a certain threshold) and therefore not too specific. After the clustering process, signatures contained in each cluster can be deployed in intrusion detection systems (IDS) such as Snort [13].

Static analysis is applied to either signature-based mechanisms or code analysis. While Pixy [10] analyses PHP code, Javascript is also a perfect breeding ground for malicious code and is therefore statically analysed by defenders, in particular to detect drive-by-download attacks.

2.2.2 Obfuscation

Although static detection methods are still actively used and useful, attackers have found the way to trick them by injecting noise and most particularly obfuscating their web-based pieces of malware.

To Xu et al. [14], there exist four types of Javascript obfuscation (cf. Table 2.1). Randomization obfuscation consists in inserting random whitespace, tabulations, carriage returns or comments. Indeed, Javascript interpreters ignore those characters. Another approach is to replace variable and function names by new randomized names. A function named `evil_function` is obviously more likely to be discovered than a the random name `f66eFkslL`.

Randomization Obfuscation	Whitespace Randomization Variable and Function Names Randomization Comments Randomization
Data Obfuscation	String Number
Encoding Obfuscation	ASCIVUnicode/Hex Coding Customized Encoding Functions Standard Encryption and Decryption
Logic Obfuscation	Insert Irrelevant Instructions Additional Conditional Branches

Table 2.1: Types of obfuscation [14]

One can also use data obfuscation. A first method is to split strings into several variables and then use the `eval` function to reassemble it (an example is shown in Figures 2.8a and 2.8b).

```
document.write('imperial')
```

(a) Not obfuscated

```
var gj = ".wr"
var bq = "'imp"
var bl = "al')"
var aw = "ment"
var fg = "docu"
var kf = "eri"
var lp = "ite("

eval(fg + aw + gj + lp + bq + kf + bl)
```

(b) Obfuscated

Figure 2.8: String splitting

Furthermore, function names can simply be replaced by new names stored in new variables (cf. Figures 2.9a and 2.9b) and a number can be written in infinite ways. For example `n = 20` is similar to `n = 10 * 2` or `n = 50 - 30`.

```
document.write('imperial')
```

(a) Not obfuscated

```
var blabla = document
blabla.write("imperial")
```

(b) Obfuscated

Figure 2.9: Keyword switching

Encoding obfuscation is also very popular among the obfuscators community. One can convert the code into escaped ASCII characters, or into a customized encoding generated by a customized encoded function. The function `eval` is then used to evaluate the real value of the code.

Finally, one can insert irrelevant functions, and modify the logic structure of the code without changing the semantics.

There even exists competitions awarding the most opaque pieces of code written [15].

Likarish et al. [16] suggested a way to detect Javascript obfuscation based on the use of classification techniques. They chose a large panel of features (detailed in section 2.3.2) to characterize a piece of malware and tested different classifiers (see section 2.3.3). They ended up with more robustness compared to rule-based techniques in that classifiers are able to detect unseen instances of malware.

Nunan et al. [9] as well worked on a classifier able to detect XSS attacks in web pages using DOM-based and URL-based features. Features and classifiers used are detailed in sections 2.3.2 and 2.3.3. They claim their work is also resistant to obfuscation although authors say it should only be used as a complimentary solution to other existing techniques.

Not all obfuscated pieces of code are malicious. That is why the technique developed by Likarish et al. cannot be employed alone to detect obfuscated malware but can be added to an existing solution to improve efficiency.

2.2.3 Dynamic analysis

An existing solution to detect obfuscated malicious pieces of code and address static analysis shortcomings is dynamic analysis.

A first instance of dynamic analysis not based on machine learning techniques is Noxes [8]: a client-side solution that detects and mitigates XSS attacks. It acts as a proxy: intercepts web pages, analyses them, extracts external and local links, and decides whether it should create a new rule to block the script or not, according to what the user decides.

An other dynamic solution is to execute potential malicious pieces of code into sandboxes and client honeypots (or honeyclients). Here, only the consequences of a piece of malware on the honeyclient matters and no longer the actual structure of it. Cova et al. [5] distinguish low-interaction honeyclients from high-interaction one. The first fake regular browsers by behaving according to predefined specifications. They are therefore limited by these specifications and can only detect certain types of malware. The latter is a sort of "super-browser" (full-featured) running in a virtual machine and keeping logs of all the modifications done to the system. If a modification is abnormal, the honeyclient triggers an alert. However, it is very difficult to create a browser that could match all the possibilities of attacks and vulnerabilities in the wild.

One of the most efficient state-of-the-art learning tool that detects drive-by-download attacks is JSAND [5]. It has the ability to be robust to obfuscation and not to be reconfigured for each new vulnerability. The detection approach lies on a large set of features ranged according to different models. A model is a set of procedures that assigns a probability score to a feature. Then all the scores are summed and form the anomaly score of the web page. A threshold is fixed and if the anomaly score is above it, JSAND triggers an anomaly.
JSAND is a dynamic tool that takes advantage of a customized browser emulated by HtmlUnit [17], a "GUI-less browser" modelling HTML documents and providing an API to work on them. HtmlUnit provides the possibility to simulate multiple and arbitrary system environments and configuration.
Eventually, JSAND classifies founded exploits and generate new signatures for detected malware.

2.2.4 Combination of static and dynamic analysis

Some approaches combine both static and dynamic analysis. For instance, Cujo [18], developed by Rieck et al. and illustrated in Figure 2.10, is a system used to detect and prevent drive-by-download attacks. It consists in a both static and dynamic Javascript analysis. Web pages are collected, transmitted into the loader and sent to both dynamic and static analysis. After detection stage, they are forwarded to the web client.

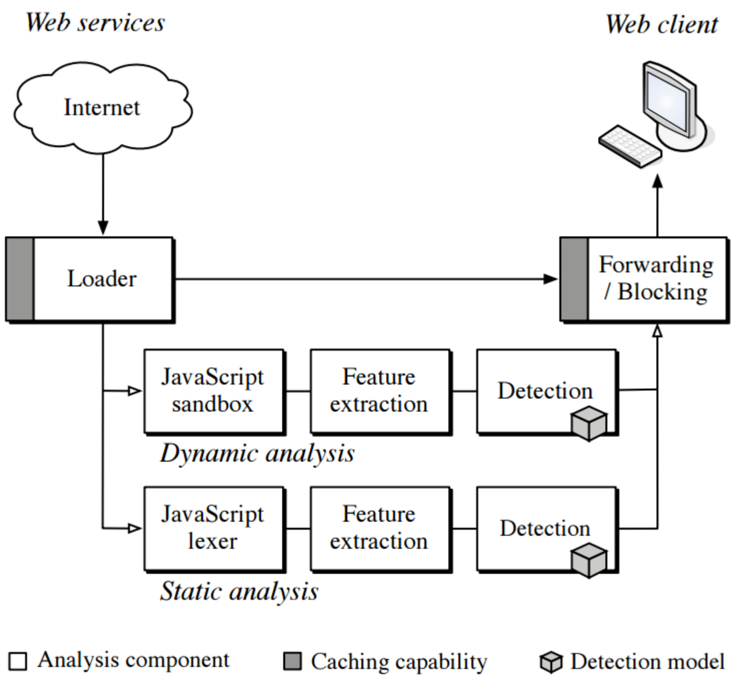


Figure 2.10: Cujo system [18]

The static analysis rests on a token extraction using a customized YACC grammar: a parser whose acronym stands for "Yet Another Compiler Compiler" [19]. The goal is, given a Javascript code, to extract a generic structure from it. Therefore, the names of all the identifiers are replaced by generic ones (e.g.: `STR`, `ID`...). Furthermore, the length of the strings used must be taken in consideration. Therefore, a string with up to 10^k characters is represented by: `STR.0`k. Note also that `eval()` function is considered as a special keyword and replaced by `EVAL`.
The dynamic analysis relies on the use of a Javascript sandbox named ADSandbox [20], and the Javascript interpreter SpiderMonkey [21]. The analysis reports all monitored operations that changed the state of the virtual browser environment on which the interpreter operates. In addition, a mechanism has been implemented to Cujo that allows it to detect patterns of common attacks such as heap spraying and re-evaluation of strings. Indeed such attacks often follow the same series of steps and are frequently used in the wild.
From static and dynamic analysis, a number of features is extracted and feeds a learning-based classifier.

2.2.5 Evasion

Obviously, there exist ways to evade such dynamic techniques. For example, an attacker can write her code in a way that bypasses the set of classifying features. Additionally, she can evade a honeyclient by detecting whether or not the emulating environment is a "real" regular one, and deciding to trigger a malicious behaviour only in the positive case: this process is known as emulation fingerprinting.

Revolver [22], by Kapravelos et al., is a tool dedicated to detecting such evasive behaviours. Assuming that we have two snippets of Javascript code that realize the same function, one is malicious and the other is benign. Then Revolver can with great accuracy detect which one is malicious. Two versions of a same script are then needed, which can be a serious limitation in the case only the malicious version is present in the wild: Revolver will not be able to detect it. But according to the authors, this case is unlikely as the majority of malware writers seek to improve an initial malware by adding evasive components and not modifying the whole structure of it. Furthermore, Revolver relies on the use of existing drive-by-download detectors that feed it with a set of malicious scripts and a set of benign ones. Therefore it is not a proper detection tool, but rather an analyzing mechanism.

Let us describe the underlying mechanisms of Revolver (Figure 2.11). First, the Oracle (the existing drive-by-download detection tool, which is Wepawet [23] here) collects the sets of both malicious and benign web pages. Then, Revolver extracts the Abstract Syntax Trees (ASTs) of the Javascript code present in these pages, turn them into normalised node sequences and applies minimisation techniques in order to gain efficiency (reducing deduplication, adding sequence summaries).

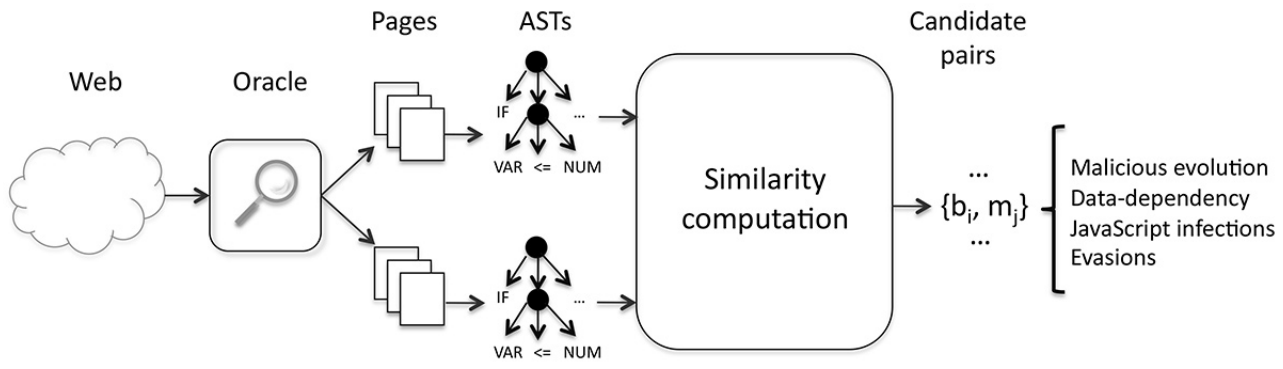


Figure 2.11: Architecture of Revolver

Next, a similarity score is computed for each pair of malicious-malicious trees and malicious-benign trees. If two ASTs or sequences of executed nodes are exactly similar, then Revolver classifies the pair as an instance of data-dependency. It can for example be the case for Javascript packers. If a malicious-malicious pair has a similarity score higher than a given threshold, this is an instance of a malicious evolution. The similarity of a malicious-benign pair can be classified along different ways: in the case of a benign sequence that would be included in a malicious one, we can conclude the attacker has injected some code to the benign one and therefore this would be classified as a Javascript injection. In the case of a malicious sequence of additional control-flow nodes included in a benign one, we have an instance of evasion. It corresponds to cases where the oracle has initially detected a malicious code, so the attacker adds components to hide its malicious behaviour and evade the Oracle. This classification is summed up in Table 2.2.

AST	**Executed nodes**	**Classification**
$=$	*	Data-dependency
*	$=$	Data-dependency
$B \subseteq M$	$\neq$	JavaScript injection
$M \subseteq B$	$\neq$	Evasion
$\neq$	$\neq$	General evolution

Table 2.2: Candidate pairs classification. B is a benign sequence, M is a malicious sequence, * indicates a wildcard value [22]

2.2.6 Different aims

Accuracy is the main characteristic a malware detector should have: it should detect malicious scripts and not flag benign scripts as malicious. However, other criteria come into play: some would want the detector to be very fast, others would focus on minimising computational costs. Schütt et al. [24] and Rieck et al. [18] decided not to create a new malware detection tool in itself but to improve existing ones to fulfill their aims: early detection and costs reduction.

2.2.6.1 Early detection

Cujo [18] is a malware detector aiming at extending web proxies. Thus, it has to be reactive and quick. Rieck et al. managed to make it act almost as fast as a regular proxy. This is illustrated by Figure 2.12 which represents the probability density of Cujo and a regular proxy in function of their run-time per URL. The run-time distribution of Cujo is slightly shifted to the right and its tail is a bit more elongated than the regular proxy's, but the global form of the distribution is utterly reasonable.

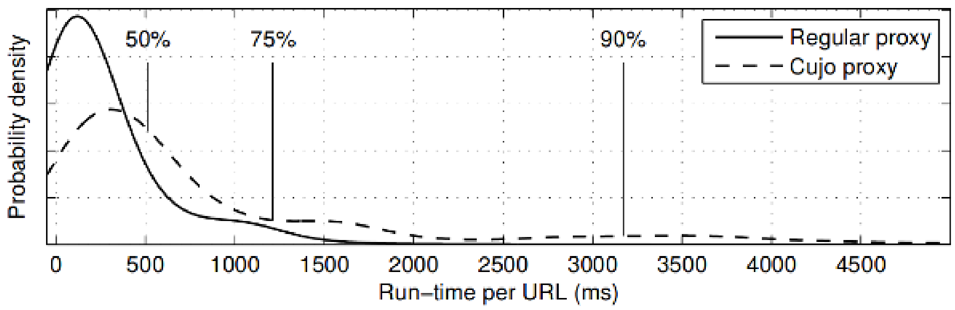


Figure 2.12: Operating run-time of Cujo and a regular web proxy on the same dataset [18]

EarlyBird [24] is a detection method created by Schütt et al. that is aimed at detecting malware as fast as possible. It extends the learning algorithm Support Vector Machines (SVM) (see 5.3) and is integrated into the Cujo detector [18] discussed above. Javascript code is monitored by Cujo at run-time and each event triggered is turned into a binary vector where the size of the vector is the number of all possible events and the i^{th} cell of the vector is either 1 if the event has been triggered or 0 if it has not. While SVM does not use temporal weighting, EarlyBird favors events that occur early in malicious code and penalises those triggered in the final execution phase of it.

Unfortunately, EarlyBird can easily be evaded. Indeed, an attacker, can pad benign-like events at the beginning of her malicious code and/or decide only to insert malicious events at the end of code. Authors claim however that EarlyBird is quite robust against this type of evasion as long as the padded events do not exceed 50% of the whole sequence.

2.2.6.2 Minimising computational costs

Large-scale analysis can be very long due to the huge number of pages to inspect. Some approaches use offline methods and thus are very effective [5] but others aim at being used in real-time. Canali et al. created Prophiler [25], a static filter quickly reducing the number of potential malicious web pages to analyse for a main detector. False negatives are critical: the filter must not discard malicious pages flagged as benign. But false positives are not problematic since all the web pages not discarded are inspected by the main detector.
Prophiler statically inspects web pages, extracting a set of HTML, Javascript and URL/host-based features. Those features then feed a learning classifier.
Although static analysis is not accurate, it is faster than dynamic methods where code has to be emulated in a virtual environment. Hence, Prophiler can quickly discard the most "obvious" web

pages. Indeed, in their experiment, Canali et al. found that Prophiler had reduced by 85.7% the load of the back-end analyzer.

2.3 Techniques and results

For the purpose of our project, we document here what methods are used in the literature and determine what are likely to be the best practices in term of data collection, choice of features, classification, validation and evaluation.

2.3.1 Data collection

To train learning classifiers and be able to detect whether a script, a piece of code or a URL is malicious, one has to collect both a dataset of benign items and a dataset of malicious examples. The underlying difficulty is to pick the items such that they are representative of real-world content, and they are correctly labeled. Table 2.3 recapitulates the datasets collected by all the authors and their origin. Note that the training dataset for each approach is taken from those collected datasets according to the validation method employed (see 2.3.4).

2.3.1.1 Benign dataset

The majority of the articles surveyed [16] [18] [5] [25] [24] chose Alexa top sites [26] as a source of benign web pages. The main reason for it is that popular websites are unlikely to trigger malicious behaviours and are supposedly very secure. This is a controversial choice and that is why some authors (for Prophiler [25] and JSAND [5]) added an extra layer of security by checking web pages crawled from Alexa with Google Safe Browsing [27] tool. Still, there are chances that it remains malicious items in the benign dataset and it is very difficult to determine with precision whether the whole dataset is perfectly benign. In addition, a manual inspection is unfeasible as we are dealing with thousands and even millions of pages.

Nunan et al. [9] decided to use *Dmoz* [28] and *ClueWeb09* [29] databases which are two sets of web pages collected and verified by their respective communities. Therefore, again, those datasets can contain a few malicious pages. Houa et al. [30] were unclear about the collection of benign pages and Perdisci et al. [11] chose to sniff the supposedly secure network of a large company. Eventually, Revolver [22] is not supposed to use benign or malicious datasets itself as it uses the Oracle ones: here more than 20 millions benign scripts are crawled by Wepawet [23].
Note that all the authors collected more than 10,000 items except for Houa et al. [30] that used less than a thousand web pages to conduct their experiments. This seems quite low and is unlikely to be representative.

2.3.1.2 Malicious dataset

Malicious items are more difficult to get. That is why their number is much lower than benign pieces of data. For example, Likarish et al. [16] collected only 62 malicious scripts compared to the millions of benign scripts they were able to get. For the majority of the articles [16] [18] [30] [5] [25] [24], the number of malicious items collected do not exceed 341, which seems quite low to represent all kinds of possible malicious behaviours. Prophiler [25] makes use of a little more data: 787 malicious web pages. Nunan et al. [9] and Likarish et al. [16] managed to collect respectively 15,366 and 25,720 distinct malicious items which looks more satisfying. Finally, Revolver [22] uses the 186,032 malicious scripts found by Wepawet Oracle.

Malware are found in both non-commercial and commercial databases and are checked manually by some of the authors [16] [18] [5] [25] [24].

Article	Malicious items		Benign items	
	Number	**Origin**	**Number**	**Origin**
Likarish et al. [16]	62 scripts	URLs blacklists [31] [32] + manual review	63 million scripts in which 50,000 are picked at random	Alexa top web sites [26]
Perdisci et al. [11]	25,720 distinct malware samples	Commercial and non-commercial malware sources [33] [34]	25.3 millions of HTTP requests	2 day sniff of a "large and well administered enterprise network"
Cujo [18]	(1) Spam Trap: 256 URLs (2) SQL Injection: 22 URLs (3) Malware Forum: 201 URLs (4) Wepawet-new: 46 URLs (5) Obfuscated: 84 URLs	Same as JSAND for (1),(2),(3),(4) (5) Additionnaly obfuscated scripts from (1), (2), (3) and (4) using a Javascript packer [35]	(1) 200,000 URLs (2) 20,283 URLs	(1) Alexa top web sites [26] (2) Insitute web surfing
Houa et al. [30]	176 DHTML web pages	*StopBadWare* blacklist [36]	965 DHTML web pages	?
JSAND [5]	(1) Spam Trap: 257 URLs (2) SQL Injection: 23 URLs (3) Malware Forum: 202 URLs (4) Wepawet-bad: 341 URLs	(1) Spam URLs provided by Spamcop [37] + local spam trap + Capture-HPC to analyze each URL + manual verification (2) Monitoring "a number of websites" (3) Forums [38] [39] (4) Wepawet	11,215 URLs	Alexa top web sites [26] + most popular queries in Google and Yahoo! search engines + Google Safe Browsing [27] to remove malicious pages
Prophiler [25]	787 web pages	*Wepawet* database + manual inspection	51,171 web pages	Alexa top web sites [26] + Google Safe Browsing API [27] to remove malicious pages
Nunan et al. [9]	15,366 web pages	*XSSed* database [40]	(1) 57,207 web pages (2) 158.847 web pages	(1) *Dmoz* database [28] (2) *ClueWeb09* database [29]
EarlyBird [24]	Same as Cujo	Same as Cujo	100,000 URLs	Alexa top web sites [26]
Revolver [22]	186,032 scripts	Output of Wepawet	20,732,766 scripts	Output of Wepawet

Table 2.3: Number and origin of malicious and benign items collected

2.3.2 Choice of features

2.3.2.1 Sets of features of the articles surveyed

All the articles surveyed except [24] and [22] use a set of features to describe and classify an item either positively (if it is a piece of malware) or negatively (if it is a benign item). EarlyBird [24] and Revolver [22] are based on existing detectors: respectively Cujo [18] and Wepawet [23] and therefore are dependent on their sets of features.

Table 2.4 sums up the number and type of features chosen for each article.

Article	**Features**
Likarish et al. [16]	65 features: 50 corresponding to Javascript keywords and symbols + 10 HTML
Cujo [18]	q-gram representation of static code and dynamical sequences of instructions
Houa et al. [30]	171 features: 154 native Javascript functions + 9 HTML + 8 advanced features such as the count of the use of each ActiveX object
JSAND [5]	10 main features in 4 groups: redirection and cloaking, deobfuscation, environment preparation, exploitation
Prophiler [25]	77 features: 19 HTML + 25 Javascript + 12 URL-based + 21 host-based
Nunan et al. [9]	6 features in 3 groups: obfuscation, suspicious patterns and HTML/-Javascript schemes
Perdisci et al. [11]	11 features: 7 statistical + 4 structural

Table 2.4: Set of features used by several approaches

All the articles refer to HTML and Javascript features which represent the core of a webpage. Houa et al. [30] added features corresponging to ActiveX objects since they are often exposed to vulnerabilities.
Note also that Cujo [18] and EarlyBird [24] make use of a q-gram representation of code and instructions and do not bother to know the actual signification of the tokens they extract. Each distinct q-gram represents a feature. One could imagine that there exists a huge number of q-grams to represent the whole script. However, as explained in section 4.4, analysed code is turned into a generic structure composed by a limited number of elements (`ID`, `STR.01` etc.). Hence, the space of all possible q-grams is relatively restricted. Therefore, the size of binary vectors representing malicious scripts is reasonable.
Eventually, Prophiler [25] and Nunan et al. [9] also refer to URL-based features.

As explained by Houa et al. [30] (see Figure 2.13), one must chose very carefully the set of features. On one hand, the more features are chosen, the more accurate the model is but the less resistant to obfuscation it is. But on the other hand, if the set of feature is too small and therefore too general, it has a good ability against obfuscation but is not accurate.

A second concern is the robustness of features: an attacker must not be able to evade the set of features to launch her attack. Likarish et al. [16] determined the most useful features for their study (detection of obfuscated code) with chi-squared statistic method. They found that features that are the most correlated with malicious scripts are: the percentage of human readable words, the use of the Javascript function `eval()`, the percentage of the script that is whitespace and the average string length and characters per line. They conclude saying that the "human readibility" feature is the main feature to be worked on in order to prevent the attacker to evade the set of features.
JSAND [5] employs a set of ten features in 4 groups (see Table 2.5). Environment preparation features (number of bytes allocated through string operations, number of likely shellcode strings)

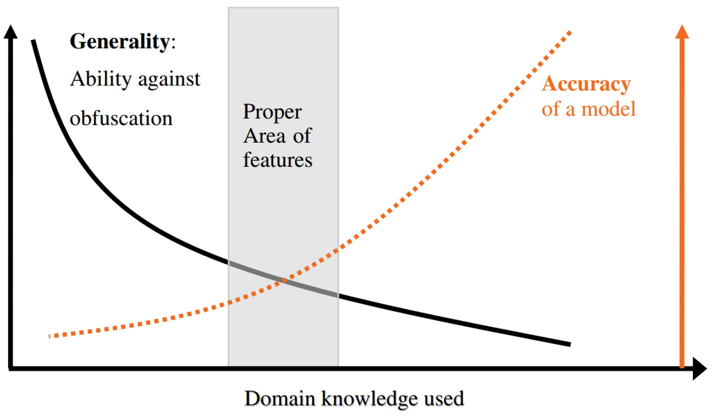


Figure 2.13: Choice of features: Obfuscation vs Accuracy [30]

and exploitation features (number of instantiated components, values of attributes and parameters in method calls, sequences of method calls) are *necessary* according to the authors in that they are required for the attacker to launch an attack. Redirection and cloaking features (number and target of redirections, browser personality and history-based differences) and deobfuscation features (ratio of string definitions and string uses, number of dynamic code executions, length of dynamically evaluated code) are *useful* features that are not always present but help the attacker to hide her malicious behaviour. Authors claim that it is very unlikely for an attacker to escape all of the four categories of features and illustrate this by showing that main attacks in the wild are based on at least two of the four categories.
To mitigate evasion, Canali et al. [25] chose to analyse a few random pages that the system classified as benign. This allows the authors to detect "systemic false negatives" and then update their feature sets and models.

Necessary	
Environment preparation	Number of bytes allocated through string operations
	Number of likely shellcode strings
Exploitation	Number of instantiated components
	Values of attributes and parameters in method calls
	Sequences of method calls

Useful	
Redirection and cloaking	Number and target of redirections
	Browser personality and history-based differences
Deobfuscation	Ratio of string definitions and string uses
	Number of dynamic code executions
	Length of dynamically evaluated code

Table 2.5: JSAND features

Furthermore, as highlighted by Canali et al. [25], benign items must not be flagged as malicious, therefore the distribution of feature values for benign items must be different than the distribution for malicious items. A good example is obfuscation: a benign web page can be obfuscated in order to prevent tampering or reverse engineering, but obfuscation is often associated with malicious pages that hide their behaviours. Consequently, not only obfuscation-based features must be used to classify an example but also others that qualify malicious behaviours. Combining several types of features prevent from getting false positives.

Perdisci et al. [11] also used two groups of features for their clustering method. Corse-grained clustering is made based on features representing statistical similarities of malware: total number of HTTP requests, number of GET and POST requests, average length of the URLs, average number of parameters in the request, average number of parameters in the requests, average response length. Fine-grained clustering considers features relative to the structure of the URL (see Figure 2.6).

2.3.2.2 Evaluation of features

Some of the articles surveyed state the most useful features for detection. They are recapitulated in Table 2.6.

Likarish et al. [16] used a chi-squared method to get features that are the most highly correlated with malicious scripts.
The most useful feature is the percentage of human readable words among the script. They define a readable word as a word which contain more than 70% alphabetical letters, between 20% and 60% vowels, the word must be less than 15 characters long and cannot contain more than 2 repetitions of the same character in a row. This can be explained by the fact attackers often obfuscate their code and do not want humans to be able to understand what they are trying to do.
The second most useful feature is the use of Javascript function `eval`. It is a common obfuscation trick: instead of displaying the name of a malicious function, or link in plaintext, attackers choose to hide it and make it the more obscure possible. Then, `eval` function evaluates the obscure piece of code and trigger the malicious behaviour.
Other useful features are the average string length and the average characters per line, which are also used during obfuscation to hide malicious content.

Rieck et al. [18] picked the top 4-grams the most useful for an obfuscated attack concerning static features and the top 3-grams concerning dynamic features. Static features show the use of `eval`, XOR and a loop, which jointly reveal the presence of a XOR-based decryption routine often used in obfuscation schemes. Among the dynamic features, `eval` function is again there, as well as `unescape`, `fromCharCode` and `parseInt` that are typical obfuscation functions found during the decryption part of an attack.

Schütt et al. [24] list top 3-grams occurring early in the execution of code and having their contribution increase, and top 3-grams occurring late in time and having their contribution decrease. The top “early” 3-grams refer to obfuscation function `unescape`. The top “late” 3-grams are features indicating the actual attack, for instance `TO HEAPSPRAYING DETECTED`. The aim of EarlyBird being to detect the attack before it is executed, this result is coherent.

In Houa et al. [30] approach, obfuscation features are as well the most useful in term of information gain: distinct word count, function name, line count and word count.

Eventually, the features the most responsible for the JSAND's anomaly score a page are exploitation features (88%) – that is to say: the number of instantiated components, the values of attributes and parameters in method calls, the sequences of method calls (see Table 2.5). This can be explained by the fact popular attacks always follow the same patterns and sequences.

Article	**Best features**
Likarish et al. [16]	Based on chi-squared statistic method, features the most highly correlated with malicious scripts: - Human readable - Use of Javascript keyword `eval` - Percentage of the script that is whitespace - Average string length - Average characters per line

Cujo [18]	Top 4-grams of an obfuscated attack for static features:
	`= ID + ID`
	`; EVAL ( ID`
	`( ID ) ^`
	`STR.01 ; FOR (`
	
	Top 3-grams of an obfuscated attack for dynamic features:
	`CALL unescape CALL`
	`CALL fromCharCode CALL`
	`CALL eval CONVERT`
	`parseInt CALL fromCharCode`

EarlyBird [24]	Top 3-grams occurring early and having their contribution increase:
	`OBJECT CALL unescape`
	`NATIVE FUNCTIONCALL unescape`
	`unescape NATIVE FUNCTION CALL`
	`CALL unescape NATIVE`
	
	Top 3-grams occurring lately and having their contribution decrease:
	`CALL substring SET`
	`TO HEAPSPRAYING DETECTED`
	`TO "..." SET`
	`TO "..." SET`

Houa et al. [30]	Based on the information gain value: - Distinct word count - Function name - Line count - Word count
JSAND [5]	Based on the share of the anomaly score of a page: - Exploitation features: 88% - Environment preparation features: 9% - Deobfuscation features: 2.7% - Cloaking features: 0.3%

Table 2.6: Set of the most useful features for detection

2.3.3 Classification

While the clustering process depicted by Perdisci et al. [11] and explained in section 4.1 is an instance of unsupervised learning, the other articles surveyed use classifiers to achieve their goals.

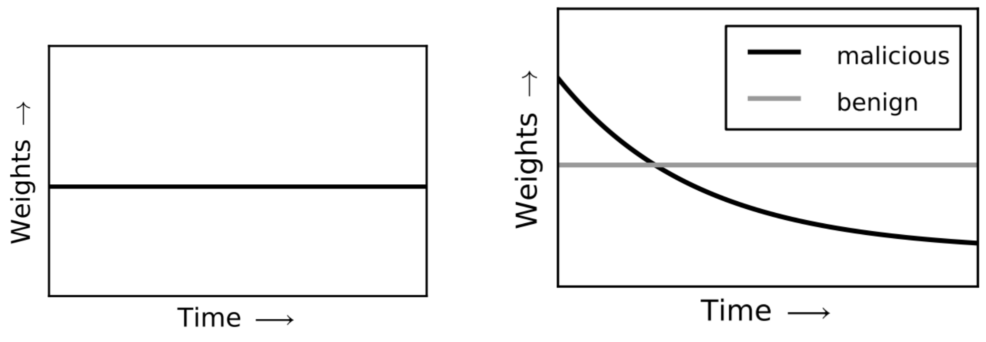
For those approaches, each item is represented by a binary vector of dimension the number of features: if the i^{th} feature is present, the i^{th} dimension of the vector has value 1, and 0 else. From this point, a classifier is used to determine if the item is positive or negative, according to the training dataset. Classifiers used for the surveyed articles are summed up in table 2.7.

Article	**Classifiers**					
	Naive Bayes	**DT & altern.**	**SVM & altern.**	**RIPPER**	**Random Forest**	**Logistic Regr.**
Likarish et al. [16]	x	x	x	x		
Cujo [18]			x			
Houa et al. [30]	x	x	x			
JSAND [5]	x					
Prophiler [25]	x	x			x	x
Nunan et al. [9]	x		x			
EarlyBird [24]			x			

Table 2.7: Learning classifiers employed in several approaches

Naive Bayes is a statistical method based on Bayes theorem. All the features are considered independent from each other. An example is classified as an instance of the class with the highest *a posteriori* probability. Naive Bayes classifiers are widely used for their low computational cost but are outperformed by other classifiers like *Boosted Decision Tree* and *Random Forest* [41].

Support Vector Machine (SVM) classifiers consist in separating the feature space by an hyperplane and therefore maximizing the distance between instances of both benign and malicious examples. While a regular SVM weighs all features the same way, EarlyBird [24] is a customized SVM classifier such that features responsible for malicious events that occur early in time have a greater weight and features responsible for malicious events occurring late in time are penalized. This is depicted in Figure 2.14.



(a) Regular SVM

(b) EarlyBird SVM

Figure 2.14: Temporal weightings for the regular SVM and EarlyBird SVM

*Decision Tree*s methods are based on the construction of a rooted tree such that each internal node of a tree is a feature, edges are values of the feature present in the previous node and node leaves are classes of the classifiers (here: positive or negative). The feature selection for a node in a tree is based on the information gain this feature has on the training dataset.

Classical decision trees are not stable, a small fluctuation in the training examples can alter classification. That is why, Houa et al. [30] chose to work on a customized *Decision Tree* classifier called *Boosted Decision Tree.* The idea here is to automatically produce a large number of small trees and to average them in order to reduce bias [42].
Canali et al. [25] use *Random Forest* which is based on tree bagging and aimed at reducing variance.
Likarish et al. [16] chose an *Alternating Decision Tree* (ADTree) classifier which combines the advantages of decision trees and boosting [43].

RIPPER is a simple rule learning system based on information gain and comprising an incremental reduced-error pruning process [44].

Canali et al. [25] also use classifiers based on *logistic regression* such as J48 or Logistic.

2.3.4 Validation and evaluation

2.3.4.1 Classifier validation

After training a classifier, one has to validate it. A number of approaches [16] [30] [25] [9] uses a 10-fold cross validation applied ten times with different configurations for the 10 folds and averaged. A 10-fold cross validation consists in splitting a dataset (see Table 2.3) in ten, take 9 out of 10 subsets to train the classifier and leave the 10^{th} subset for validation. Others [18] [24] decided to split their collected dataset and take 75% of it for training and 25% for validation. Statistics computed from the validation step such are recapitulated in Table 2.8

Name	**Formula**	**Signification**
Precision or PPP (Positive Predictive Power)	$\frac{TP}{TP+FP}$	Number of malicious URLs actually truly malicious among all the URLs found as malicious
Recall	$\frac{TP}{TP+FN}$	Number of malicious URLs actually truly malicious among all the true malicious URLs
F_2 score	$\frac{5 \cdot Precision \cdot Recall}{4 \cdot Precision + Recall}$	Weighted average of precision and recall
NPP (Negative Predictive Power)	$\frac{TN}{TN+FN}$	Number of benign URLs actually truly benign among all the URLs found as benign
Accuracy	$\frac{TP+TN}{FN+TN+TP+FP}$	Number of malicious and benign pages that have been well labeled among all the pages
Specificity	$\frac{TN}{TN+FP}$	Number of benign URLs actually truly benign among all the true benign URLs

Table 2.8: Validation metrics

TP, TN, FP, FN are defined such as:

- TP is the number of true positive items, ie: the number of malicious items labeled correctly.
- TN is the number of true negative items, ie: the number of benign items labeled correctly.
- FP is the number of false positive items, ie: the number of benign items flagged as malicious.
- FN is the number of false negative items, ie: the number of malicious items flagged as benign.

Four of the articles surveyed use several classifiers and had then to compare their performance. Likarish et al. [16] found that, among Naive Bayes, ADTree, SVM and RIPPER, the classifier with the highest precision on the validation set is SVM, but best recall and F2 scores are attributed to RIPPER.
Nunan et al. [9] report very close performance for Naive Bayes and SVM, with a little advantage of SVM for detection, accuracy and false alarm rate. However, as Naive Bayes has very good results too, they claim it would be preferable to use it due to its lower computational costs compared to SVM.
Houa et al. [30] compare statistics of Decision Tree, Naive Bayes, SVM and Boosted Decision Tree classifiers. They find that Boosted Decision Tree achieves the best performance in term of accuracy and false positive rate.
Canali et al. [25] trained their classifiers on different sets of features and compared false negative and false positive rates. They conclude that Random Forest is the best classifier for HTML features, and J48 is the best classifier for both Javascript and URL+host features. Prophiler has then be configured with those classifiers.

2.3.4.2 Global results of the articles surveyed and discussion

Rieck et al. [18] computed 94.4% of true positive items on the attack datasets and only 0.002% of false positive items on the benign datasets. False alarms were caused by fully encrypted and obfuscated Javascript that looks very much like malicious scripts. Nunan et al. [9] find detection rates higher that 94% on their malware datasets for each of their classifiers. Accuracy rate is above 98.54% and false alarms do not exceed 0.51%. Houa et al. [30] achieve an accuracy of 96% on their validation dataset. Boosted Decision Tree also classifies only 0.21% of false positives.
These are apparently very good results, but would need to be confirmed by an extra larger set of validation, or a real-world deployment experience.

After a primary validation, some authors decided to apply their four classifiers to an other testing dataset. Likarish et al. [16] crawled 24,269 scripts from blacklisted domains [45]. Each of the classifier found around 20 malicious examples and after manual inspection, no more than 5 examples were false positives, except for RIPPER classifier which classified 9 items out of 28 wrongly. However, false negatives have not been computed for this experiment.

Canali et al. [25] used an extra validation set of 153,115 pages transmitted and labeled by Wepawet. That allowed the authors to compare the results found by Prophiler and Wepawet. Prophiler was able to discard 124,906 pages producing only 0.54% of false negatives and 10.4% of false positives, which is not a problem knowing that Prophiler is only a filter and not a malware detector on its own. After that, Canali et al. decided to perform a large-scale evaluation on 18,939,908 pages in order to measure run-time performance. Prophiler has reduced the load on the back-end analyzer of 85.7% and has achieved its goal of filter. Note here that the reference is Wepawet: false positives and false negatives are computed from this tool but we do not know whether the figures are right in absolute. The critical point here is false negatives. Authors could not manually inspect all the pages crawled but decided to randomly choose 1% of the benign-labeled pages to check whether they could find malicious examples among them. They finally discovered 3 malicious items out of 162,315 pages.

Cova et al. [5] divided their benign dataset in three subsets: the first part for training, the second for the establishment of an anomaly score threshold and the third for false positives computing (there were no false positives on this set). They also computed the false positive rate on an other dataset composed of 115,706 URLs crawled on Google, Yahoo! and Live search engines. They inspected the dataset manually and found 122 out of 137 URLs flagged by JSAND were actually malicious. In addition, the relatively small number of malicious samples collected through several malware databases (see Table 2.3) allowed the authors to manually inspect all of them in order to compute the false negatives. They applied JSAND and other detection tools (ClamAV [46], PhoneyC [47], Capture-HPC [48]) to this malicious dataset and found that JSAND had the best results with a total of only two false negatives.

EarlyBird [24] is an extension of Cujo detector. The same configuration as the work of Rieck et al. [18] has been used except for the classifier: EarlyBird customized SVM has replaced the regular SVM. EarlyBird detector found 93.2% of true positive items while the basic Cujo detector found 90.1%. EarlyBird also outperforms regular Cujo in terms of false positive rate with 0.005% detected instead of 0.01%. As expected, Schütt et al. managed to implement a tool capable of detecting malicious code before the regular Cujo detector: an average of 43.6% of code is executed before flagging a script as malicious, while the basic detector executes 70.5% of it.

Revolver [22] flagged 155 pairs of ASTs as instances of evasion. Authors manually inspected all of them and found that only five of them were false alarms. The drawback of this method is that every pair labeled as evasive must be manually inspected to be validated. Authors claim they plan to build automatic tools that can confirm similarities without needing manual review.

To evaluate the signature generated by their clustering method, Perdisci et al. [11] followed several steps. First, they generated sets of signatures for 6 consecutive months (from February 2009 to July 2009). Then they compared the signatures they found with a one-day legitimate traffic of a large company to discard false positive items: 'benign" signatures are pruned. Next, they tested the signatures extracted from a given month to malware samples collected in the same month and future months. Results are shown in Table 2.9. *Sig_Feb09*, *Sig_Mar09* etc. represent pruned signatures sets of months *Feb09*, *Mar09* etc. The clustering method is able to detect no less than 58.9% and up to 85.9% of malware samples collected the same month of signature generation. Obviously, when applied to a future malware datasets, detection rate becomes less accurate. Indeed, for a signature set applied to malware samples collected one month later, the detector is only able to detect from 26.4% to 50.4%.

	Feb09	*Mar09*	*Apr09*	*May09*	*Jun09*	*Jul09*
Sig_Feb09	85.9%	50.4%	47.8%	27.0%	21.7%	23.8%
Sig_Mar09	-	64.2%	38.1%	25.6%	23.3%	28.6%
Sig_Apr09	-	-	63.1%	26.4%	27.6%	21.6%
Sig_May09	-	-	-	59.5%	46.7%	42.5%
Sig_Jun09	-	-	-	-	58.9%	38.5%
Sig_Jul09	-	-	-	-	-	65.1%

Table 2.9: Signature detection rate on current and future malware samples (1 month training) [11]

Perdisci et al. finally computed false positives by comparing each month the set of pruned signatures to a second day of legitimate traffic in the same large company. They found no more than $3 \cdot 10^{-4}$ items wrongly flagged as malicious (on March 2009).
Eventually, they applied their clustering method to a real-world experience, by monitoring the traffic of a large company for four days and applying the pruned signatures they got previously. They managed to find malicious signatures from 46 machines that were not detected by antivruses.

2.3.4.3 Validity analysis

Most of the papers surveyed compare their results with existing detectors.

Cova et al. [5] compared JSAND's false negative rate with ClamAV [46], PhoneyC [47] and Capture-HPC [48], on their malicious datasets. JSAND (0.2% of false negative) largely outperforms ClamAV (80.6%) and PhoneyC (69.9%) and is also better than Capture-HPC (5.2%). The three tools missed a large number of malicious items whereas JSAND only missed 2 of them.

Rieck et al. [18] compared Cujo with JSAND [5], ClamAV and AntiVir on the malicious datasets. As JSAND is an offline tool, this is not a surprise to see that it achieves a higher true positive rate (99.8% instead of 90.2%). Cujo still outperforms AntiVir and ClamAV which have respectively true positive rates of 70% and 35%. It is difficult to perform a representative comparison between JSAND and Cujo is term of false positive as they do not use the same sets of benign items. However when applied on the same benign datasets as Cujo's, ClamAV had a false positive rate of 0% whereas Cujo reaches 0.002% on its 'Alexa-200k" dataset. On this latter dataset, AntiVir computed a false positive rate of 0.087%.

The extended version of Cujo, EarlyBird [24] was confronted to a regular Cujo occurring over time, one launched at the end of code execution and AVG Anti-Virus [49] launched as well at the end of code execution. The "over-time" Cujo achieved 90.1% of true positive, the "end" Cujo 88.0% and AVG obtained 91.8%. EarlyBird had a better rate with 93.2%. Note that measures of false positive rates were similar to all the methods.

A comparison of Prophiler [25] with the work of Seifert et al. [50], Ma et al. [51], Feinstein et al. [52] and Likarish et al. [16] showed that this technique has lower false positive an false negative rates than all other past approaches cited. Indeed Prophiler managed to get 9.88% of false positive and 0.77% of false negative whereas other approaches got between 13.70% and 17.09% of false positive and between 2.84% and 14.69% of false negative. Note that Canali et al. chose a same comparison dataset to apply those different methods.

Houa et al. [30] applied different anti-virus software to their training dataset, denoted by two letters: AA, SA, NO, TR and KA. Their learning approach is much better than other antivruses: their true positive rate is more than 85% whereas the best antivirus they use has a true positive rate of less than 50%. False positive are very low in every cases. Authors underline that it might not be a fair comparison because they applied their method to the training dataset, so there is no guarantee that their approach would detect types of malware that are new and not present in their dataset.

Nunan et al. [9] compared their work with Likarish et al. [16] detector. They use SVM classifier on their malicious datasets. While Likarish et al. get a true positive rate of 91.3%, Nunan et al. achieve 94.0%. NPP is also better with 99.4% compared to Likarish et al.'s rate of 99.1%.

2.3.5 Best practices

In the light of all the techniques used and the results obtained in the literature and in the wild, we determine what are likely to be the best practices. They are recapitulated in Table 2.10.

Data Collection	Benign Dataset	- Alexa top websites - Google Safe Browsing - Manual inspection of a few random pages - more than 10,000 items
	Malicious Dataset	- Blacklists - Social media: malware hunters, researchers - Demo Website - the largest amount possible
Features	Meaningful	- Obfuscation - Exploitation - Keywords
	Non-meaningful	- q-grams
Classification		Compare several classifiers included: - Naive Bayes - SVM - Decision Tree
Validation		Use several metrics to compare cross validations: - Precision - Recall - F_2 score - NPP - Specificity - Accuracy
False Positive and False Negative		Manual inspection of a few random pages
Validity Analysis		Compare the detector with those found in the literature and with popular antiviruses

Table 2.10: Best practices found in the literature

For data collection, collecting data from popular websites seem to be the best option. A further layer of security can be added to this set by applying Google Safe Browsing [27]. In order to detect systemic errors, one can inspect a few random pages manually. From the articles surveyed, it is enough to get 10,000 items for the dataset to be both representative and large enough.

Malicious samples are found via blacklists, social media such as malware hunters or researchers. In practical, it is difficult to get many of them since as soon as they are found and shared in the community, websites are quickly taken down and cleaned. The challenge is here to inspect a contaminated webpage as quick as possible in order to get the wanted information.

Another approach to enlarge the training dataset or the validation dataset of our machine learning process is to create a demo website on which an exploit kit is set up. The difficulty here is to get an "up-to-date" exploit kit in the wild, possibly without having to pay for it...

There are two different approaches to the selection of features. We can either choose to extract meaningful features or q-grams. For the first case, we focus on obfuscation and exploitation metrics, as well as typically malicious keywords. For the latter one, we have to find a way to apply a sliding window on the pages to get the q-grams, and store them into a relevant data structure so that the high feature dimension does not impact the performance of the detector.

The collected data and extracted features feed then several classifiers. Typical and relevant ones are Naive Bayes, Support Vector Machine (SVM) and Decision Tree.

Those classifiers need to be compared over the same metrics. A good approach is to use cross validation, that is to say splitting the dataset in n parts, using all $n-1$ parts as the training dataset and one part as the validation dataset. We have then n sets of values for the different metrics (precision, recall, F_2 score, NPP, specificity and accuracy) that we average.

False positive and false negative examples found in real world deployment must be analysed in order to correct eventual systemic errors. The manual inspection of a few random pages is a good practice.

Finally, a validity analysis is also welcome in order to compare the results with other approaches and discuss the results.

Chapter 3

Methodology

The aim of the project is to build a detector of malicious web content based on the best practices collected in the literature and recapitulated in section 2.3.5. We describe here the methodology used to create such a detector. A global view is depicted by Figure 3.1.

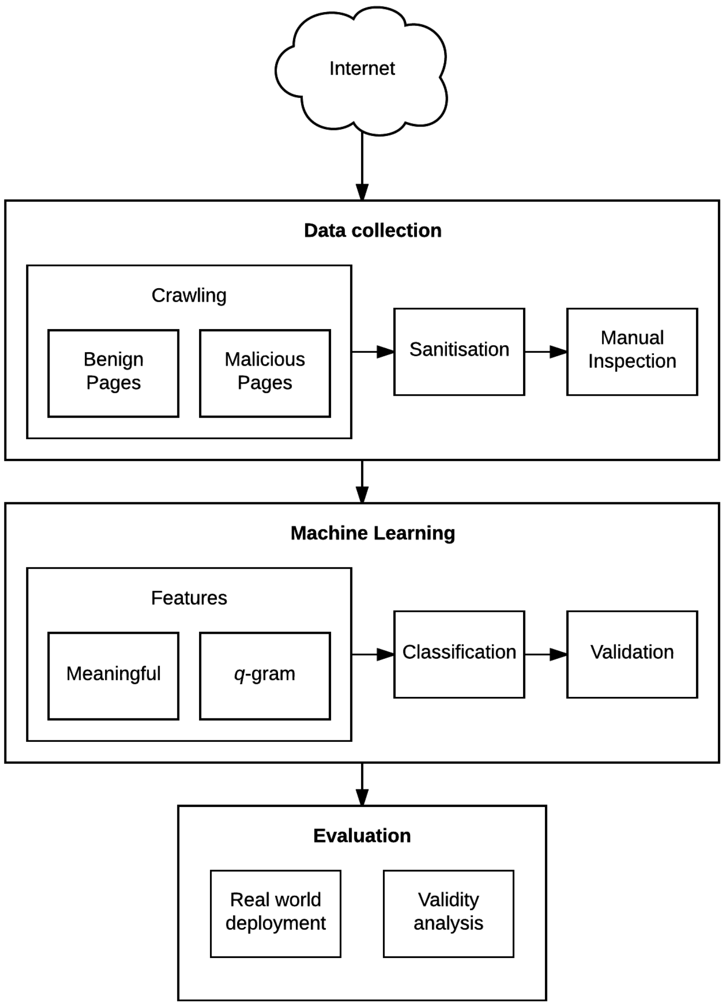


Figure 3.1: Global view of the project

3.1 Data collection

As explained in section 2.3.1, a critical point is to collect representative and accurate data for both benign and malicious dataset. We therefore need to crawl a number of URLs for both datasets, sanitise them and inspect manually part of them as a validation.

3.1.1 Benign dataset

To constitute the benign dataset, we consider the 500 most popular websites from Alexa's list. Alexa is a company founded in 1996, acquired by Amazon in 1999, and providing data analytics on the internet. In particular, they take an inventory of page views and unique site visitors and then compute a popularity score for each webpage. They end up every three month with a list of the most popular domains on the web [53].

Picking the most popular websites as the benign dataset is controversial. This choice is based on the following hypothesis: the more the website is popular, the more likely it is secure and the less likely it is exposed to vulnerabilities. Indeed, we can expect the huge companies responsible for those websites to be aware of security issues and to have hired security experts at some point. Judging by the security breaches impacting large companies like Linkedin, eBay or MySpace [54] in the last ten years, this hypothesis may not entirely hold. However this source of URLs is less likely to contain malicious content than less popular websites.

Still, this supposedly benign dataset must be sanitised and most obvious malicious webpages must be spotted and removed. To do this, we can remove crawled URLs contained in Google Safe Browsing's blacklist [27]. Furthermore, this is impossible to inspect the thousands of pages collected manually. But picking a few random pages and analysing them manually reports systemic errors and allows to have an idea of the noise in the dataset.

3.1.2 Malicious dataset

It is more difficult to collect malicious items that are still used in the wild. A number of blacklists exists like MalwareDomains [45] and MalwareDomainList [38]. They list dozens of malware landing websites. Researchers and so-called malware hunters also regularly post links pointing to malicious webpages.

We must make sure that the items collected are really malicious. Usually, when a website is infected, it is taken down by the owner as soon as possible in order not the malware to be spreaded. To overtake this issue, we create in this project a automated tool to sanitise the malicious dataset. This tool simply looks at the collected pages and determines whether a webpage is still accessible "normally" and does not lead to an error code (e.g. 403, 404, 502, 503).

Similarly to the benign dataset, inspecting manually a few random pages allows to quantify systemic errors and noise.

3.2 Extraction of features

We compare two different types of features (see section 2.3.5) for our experiment. The so-called meaningful features – features that have a real meaning, and non-meaningful features. Among those two types of features, we distinguish static features from dynamic features and we use different approaches to extract them.

3.2.1 Meaningful features

Dynamic features are extracted via the use of honeyclients allowing us to fake a regular browser, record events and find malicious behaviours that slipped through the net of static analysis with obfuscation. We use Selenium [55] as our "superbrowser", and especially its WebDriver API. The API is accessible in all the popular languages, including Python which is the language that we use for our detector. A more detailed description of Selenium is found in section 4.1.3.

Although no dynamic feature is actually extracted by our detector (lack of time), we made the choice to extract static ones from HTML and Javascript files also via Selenium. The extraction is this way done by one tool and we are sure that HTML and Javascript that we get are the same as a regular user would get.

We applied a Python's built-in HTML parser to get every Javascript file and snippet into the given webpage. Metrics are then computed thanks to basic functions (letter, word, line, keyword counts). A list of the features we use in the detector is given in section 5.2.

3.2.2 *q*-grams

q-grams are extracted from the webpages after a few steps summed up in Figure 3.2. First, we turn the code into a sequence of generic tokens in order to normalise the code. For example, in a Javascript code, we turn all the strings into a generic token called `STR`. Secondly, we create a sliding window of length q (we try several values for q) that represent one q-gram. Unfortunately, we did not have time to implement this method.

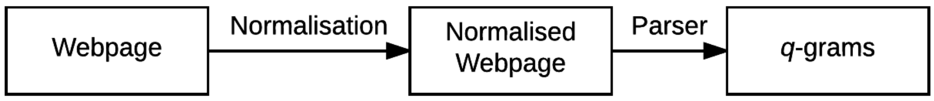


Figure 3.2: Turning a webpage into a q-grams

3.2.3 Choice of features

Once our features are extracted, feed the classifier and that we get the corresponding results, a good approach is to modify the set of features to determine whether we get better results. Of course, it is not possible to test every combination of features, it would take too much time. However, different sets can be tested after analysing the results of each experiments.

3.2.4 Machine Learning

It is very interesting to be able to compare the results of diverse classifiers on our datasets. The Python's package `sklearn` allows us to try a large panel of them. It also allows to perform cross validation on our dataset (see Figure 3.3).

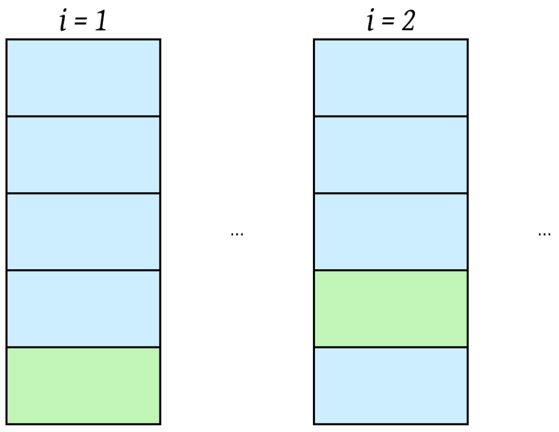


Figure 3.3: 5-cross validation. For each round of the cross validation $i \in [\![1, 5]\!]$, the dataset is split into 4 training parts (blue) and one validation part (green)

We chose to split our set in five parts. For each round of the cross validation, one part is used as the validation set and the four others are left for training.

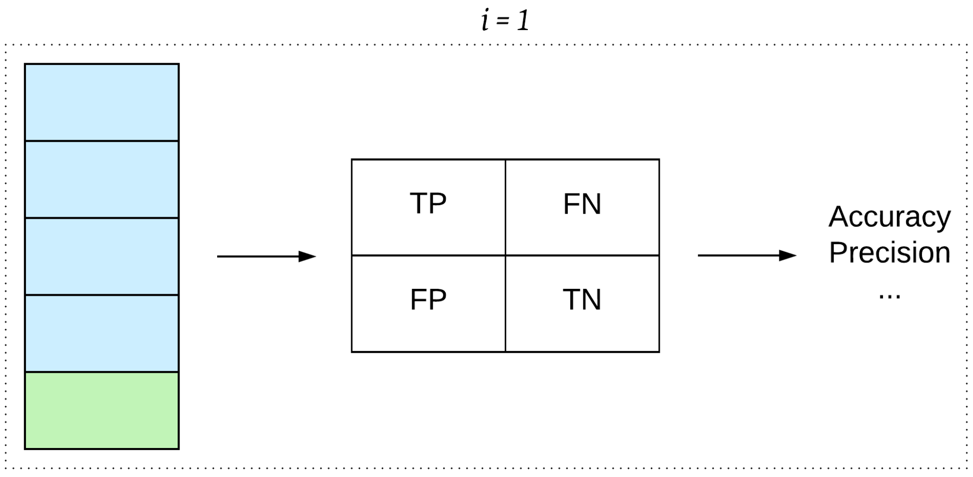


Figure 3.4: Computation of validation metrics in one round of the cross validation

From that (see Figure 3.4), a confusion matrix is computed and gives the values of true positive (TP), true negative (TN), false positive (FP) and false negative (FN). From those four values, a bunch of validation metrics are computed (precision, recall, F_2 score, NPP, specicity and accuracy) according to the formulas given in Table 2.8. After the five rounds, we get five different values for each of the validation metrics. The final result is the average of those five values.

Note that we use four classifiers: SVM (Support Vector Machine), Decision Tree, Random Forest and AdaBoost to classify our data. Given that Decision Tree is based on features gain, it allows us to access to those gains and then find the most useful features.

3.2.5 Evaluation

Once the classifiers are trained and validated, we want to try the detector in the "real world". To do that, we run a web server on which our classifier is accessible and waits to be called. It is accessible via the web server URL. Implementation and details are explained in section 4.3.

Furthermore, we created a Firefox extension which, each time a user visits a webpage, sends the webpage to our detector stored in the webserver and gets the prediciton result from it. Again, more details can be found in section 4.3.

Finally, we wanted to know whether from a certain benign URL an user has the possibility to click on a malicious URL. More generally, it is interesting to know how many websites are likely to be malicious if an user clicks on n links from the first benign website. To do that we use a spider: a tool that grabs all the URLs from a certain webpage. We then predict the label of each of the URLs and apply again the spider until the wanted depth is reached. A basic approach is to draw a tree of the form of Figure 3.5.

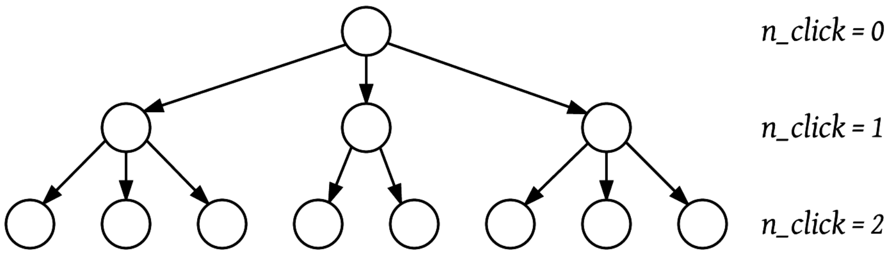


Figure 3.5: Tree of URLs visited *n_click* clicks from the first one

Very often pages accessed from a previous website refer back to this website. This would either create loops in our tree or use our classifier more than need. A more effective approach is then to create an oriented graph where each node would be an URL that have been found applying the spider n times, and each of the oriented edges refers to the presence of the URL of the destination node into the webpage accessible via the URL of the source node. This is depicted in Figure 3.6.

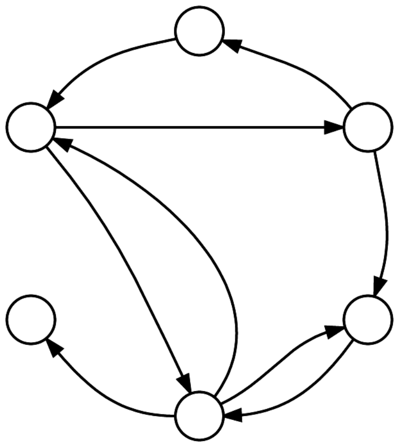


Figure 3.6: Oriented graph of visited URLs

Chapter 4

Implementation

After depicting the methodology used to build the detector, represented by Figure 3.1, let us focus on its implementation. We first recapitulate the tools we need to create our detector. Next, we explain the core program used to train and validate the classifier. Then, we show how we design our web server and the browser extension communicating with it. Finally, we take a closer look on the implementation of the tree introduced in section 3.2.5.

4.1 Implementation choices

The project consists in creating a detector of malicious web pages using machine learning techniques. We have decided to implement our work in Python at all the levels: from data collection to features extraction, classification and prediction. Python is indeed a simple and quick language which also implements all the APIs of the tools we use.

4.1.1 Requesting web pages

Several factors are at stake in requesting and getting the web pages on the internet. First of all, we must get accurate data to be able to correctly extract the expected metrics. Secondly, we deal with a huge number of pages (several thousands) so we must get the pages quickly. However, the detector must access the pages like a human would do it, that is to say, not that quickly. Else, malware could fingerprint non-human users and evade the detector system. Therefore, our malicious pages detector must remain undetectable and not become the biter bit.

A first naive solution in the use of Python's `urllib2` module [56]. A more advanced and dynamic solution is Selenium [57].

4.1.2 `urllib2` module

`urllib2` module is used to access a webpage without opening a browser. Metadata options can be added, especially the option allowing to add a user-agent to the request. Indeed, some websites block pages with unknown or none user-agent. They can either check it by inserting a script in their pages, a command in the .htaccess file, or on the server side. The advantage of this method is its rapidity in collecting the different web pages. But the drawback is that it certainly does not simulate an genuine user visit on a webpage: both for the time spent on it and the lack of genuine metadata.

4.1.3 Selenium

To make the connection of the detector genuine-ish, there is nothing better than an automated web browser, like Selenium [57]. Basically, this tool is used by web developers to automatically test their website and find bugs that they could not see manually. We decide here to use it for another purpose: dynamically accessing and storing webpages and faking the behaviour of a real user.

4.1.3.1 Selenium WebDriver

Selenium WebDriver [58] allows the user to test a webpage on a number of browsers (Firefox, Chrome, Internet Explorer, PhantomJS, HtmlUnit etc.). When a test is run, the browser opens and the test is done in "real-user time". In addition, the tool allows us to get a bunch of interesting metrics both static and dynamic. Although we can get relevant measures and accurate webpages, this method is time-consuming and cannot fit with huge datasets.

Selenium WebDriver API is available in Python by downloading the module `selenium`. For the project, we accessed all the webpages using the Firefox WebDriver which can be instantiated using the method `webdriver.Firefox()` in the `selenium` module.

4.1.3.2 Selenium Grid

It is possible to reduce the scraping time by paralleling the webpages accesses by the WebDriver. This is what Selenium Grid has been created for. It basically consists in running different tests on several machines with same or different browsers. One has to configure a machine as a hub and the other computers as nodes. The hub then dispatches the test among the nodes.

4.1.4 Classification

We use the Python's package `sklearn` [59] to implement the machine learning part of the project. It is easy to use and allows to choose a large selection of classifiers [60]. In particular, it is very easy to use SVM (Support Vector Machine) [61], to train, cross validate the classifier and get a prediction on a new sample, as shown in Figure 4.1

```python
from slearn import svm
X = [[3, 3], [4, 4]]
y = [0, 1]
clf = svm.SVC() # instantiation
clf.fit(X, y) # training
clf.predict([[2, 2]]) # prediction
```

Figure 4.1: Basic example of `sklearn`'s SVM classifier

4.1.5 Storing experiments

The solution chosen to store web pages and features values is the NoSQL open-source document database MongoDB [62]. Indeed, it guarantees:

- high performance: which is essential as our datasets are large.
- high availability: which is crucial if we want our detector to be running continuously.
- automatic scaling: which is important if we want to add fields or features to our documents.

Plus, the dynamic queries supported by MongoDB are almost as powerful as SQL's and documents are stored as JSON files.

4.2 Training and validation

Figure 4.2 shows the codebase architecture of the main program that trains and validates the machine learning classifier.

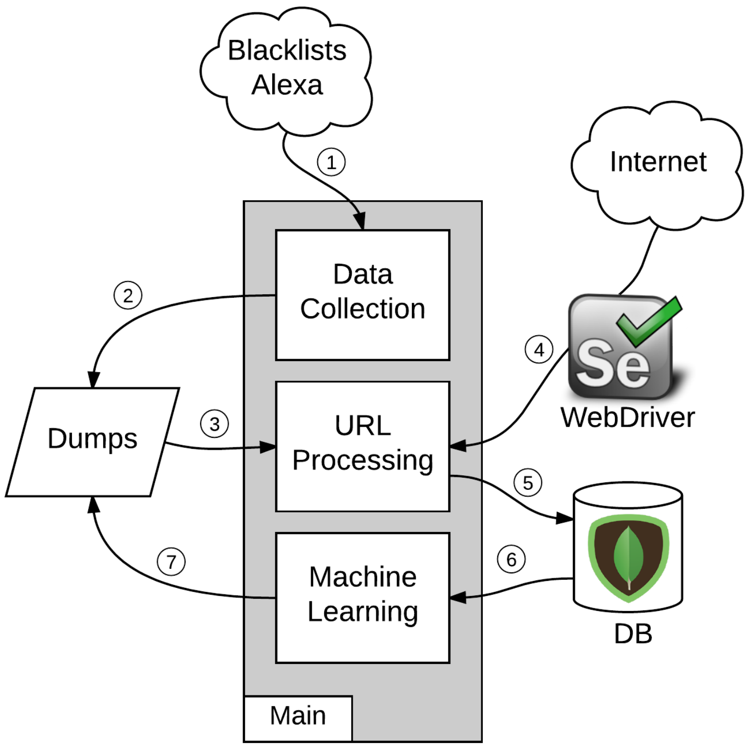


Figure 4.2: Architecture of main program

It is split into three different entities: Data Collection, URL Processing and Machine Learning. Those three entities interact with external elements: data dumps (text files and object dumps from Python), blacklists (Malware Domain List [38] and Malware Domains [45]) and a benign list (Alexa [26]) found on the internet, web pages crawled on the web, Selenium WebDriver (see section 4.1.3.1) and our database (MongoDB). These interactions are recapitulated below:

Step 1:

The functions `malicious_crawl()` and `benign_crawl()` from the Data Collection entity visit blacklists and benign list on the internet.

Malicious websites are found on MalwareDomains [45] and MalwareDomainList [38] blacklists. The first one regularly provides up-to-date text lists of malicious URLs. Therefore, we created a script accessing this list and getting the most recent URLs (according to the date attributed to each URL in the text file). The second blacklist provides an RSS feed giving the latest malicious URLs found. A simple parser allows to get those URLs. Both sources let us access the metadata such as IP, malicious source (first place where the website has been labeled malicious), malicious type (whether the website has been targeted by an exploit kit, a cross-site scripting attack etc.) and the date where the URL has been found malicious.

Alexa website [26] lists the most popular websites for each country, in particular for our case:

in the United Kingdom. They provide a charged API that allows the user to get the ranking of millions of pages. They also provide a list of the 500 most popular websites for free. Because we preferred not to pay for the API, we simply wrote a script that fetches the top 500 URLs from their webpage.

Step 2:

Malicious and benign URLs are written into two different text files (one URL per line). Those text files are stored in the Dumps folder.

Step 3:

The URL Processing entity opens the two files where both benign and malicious URLs are stored and read them line per line (one line of a file corresponding to one URL and its metadata).

Step 4 and 5:

Whether it is benign or malicious, the URL processing entity accesses the URLs from the text files with Selenium WebDriver. In particular, we apply Firefox WebDriver. Therefore, for each of the URLs, Selenium opens the Firefox browser and visits the wanted page.

We then, through the intermediary of Selenium, save both HTML and Javascript files and snippets collected while visiting the webpage into Python variables. Simple Python functions allow then to extract the features: letter, word, line and keywords counts.

The processed URL is then stored into the database. Figure 4.3 shows the generic form of a malicious URL stored into the database. Let us recall that we use the NoSQL solution MongoDB that stores the items into a collection, which has in fact the structure of a JSON file. Benign items are similar but do not possess the specific fields `malicious_src` and `malicious_type`.

```
{  "_id": /* MongoDB ID */,
   "url": /* URL name */,
   "ip": /* IP address */,
   "page_b64": /* Page encoded in base64 */,
   "added_date": /* Added date */,
   "last_modified": /* Last modified date */,
   "type": /* Benign or Malicious */,
   "description": /* Text description */,
   "malicious_src": /* Where the URL has been first referenced */,
   "malicious_type": /* Type of maliciousness (eg: EK) */,

   "method": /* Scraping method (eg: Selenium) */,
   "user_agent": /* User Agent (eg: Firefox) */,
   "code":/* HTTP status code (eg: 200, 404, 503) */,
   "static_features":[
      {
         "html_letter_count": /* Integer */,
         "html_line_count": /* Integer */,
         "html_word_count": /* Integer */,
         "js_max_letter_count": /* Integer */,
         "js_max_line_count": /* Integer */,
         "js_max_word_count": /* Integer */,
         "js_min_letter_count": /* Integer */,
         "js_min_line_count": /* Integer */,
         "js_min_word_count": /* Integer */,
         "js_scripts_count": /* Integer */,
         "js_total_escape_count": /* Integer */,
         "js_total_eval_count": /* Integer */,
         "js_total_letter_count": /* Integer */,
         "js_total_line_count": /* Integer */,
         "js_total_unescape_count": /* Integer */,
         "js_total_word_count": /* Integer */,
         "js_total_whitespace_count": /* Integer */,
         "js_min_letter_by_word": /* Integer */,
         "js_max_letter_by_word": /* Integer */,
         "js_total_letter_by_word": /* Integer */,
         "js_min_word_by_line": /* Integer */,
         "js_max_word_by_line": /* Integer */,
         "js_total_word_by_line": /* Integer */
      }
   ],
   "dynamic_features":[{}],

   "stat_crawling_time": /* Crawling time in seconds */,
}
```

Figure 4.3: Malicious URL into the database

The case explained above is valid only if the URL is not present in the database. Other conditions apply if the page is already stored. Figure 4.4 shows the pseudo-code of the main function of the entity URL Processing.

After having selected the method and user-agent we use to crawl the webpages, we save both the URLs stored in the text files from Dumps and the URLs already stored into the database in respectively `urls_to_analyse` and `urls_list`.

For each URL in `urls_to_analyse`, we check if it is already stored in the database, if we must compute new features, if the method and user-agent of the page stored in the database is the same as new one. Depending on the results of those checks, different actions are triggered: from adding the new URL, to updating it with the good parameters.

```
- Select METHOD and User-Agent
- urls_to_analyse = URLs from the list (Alexa or blacklists)
- urls_list = URLs already stored in the db

for u in urls_to_analyse:
    - Set a timeout
    - try: If timeout not exceeded
     - | if u in urls_list:
       | - | if there is new features to add:
       |   |   | if METHOD or User-Agent different than those in the db:
       |   |   | - Update URL with reloaded webpage with METHOD, UA
       |   |   |   and new features to add
       |   |   | else: Update URL with webpage stored in the db and new
       |   |   |       features to add
       |   | else: URL already stored and not modified.
       | else: Add URL with METHOD and UA.
    - except: End the process if timeout exceeded
```

Figure 4.4: Pseudo-code of URL Processing's main function

Step 6:

Once the URLs are stored into the database, the Machine Learning entity accesses it and turns the huge JSON file into two vectors X and y. Let n_feat be the number of features stored for an URL and n_urls the number of URLs stored in the database. X and y have length n_urls. The i^{th} cell of X, where $i \in [1, n\_urls]$, contains a list of the features values of the i^{th} URL stored into the database. This list is hence of length n_feat. The i^{th} cell of y contains the label of the i^{th} URL, that is to say: 0 if the URL is benign or 1 if malicious.

The next step is to instantiate the classifiers – in our case SVM, Decision Tree, Random Forest and AdaBoost. It takes only one line thanks to the `sklearn` Python's package. To train the classifiers, one only has to call their inner method `fit()` and give X and y as arguments.

The Machine Learning entity also performs the cross validation via `cross_validation_score()`. Given X, y and a classifier, the function returns a bunch of validation metrics: precision, recall, F_2 score, NPP, specicity and accuracy.

In our function we had the choice either to use the `sklearn` function `cross_val_score()` or `cross_val_predict()`. The first one performs the cross validation and returns a list of predefined metrics whereas the second returns a vector y_pred of the "averaged" predictions of each of the items after the cross validation. Although the first method is quicker, we chose the second option which is more general and allows to get more metrics.

From y and y_pred, we compute the confusion matrix (again, `sklearn` does all for us). And from this confusion matrix, we calculate the wanted metrics.

Step 7:

The fitted classifiers, X and y are saved as dump data in Dumps so that they can be reused for prediction.

4.3 Online detector

After the training part, let us move to the evaluation part. We describe in this section how the detector can be practically used to predict whether an URL is more likely to be malicious or not.

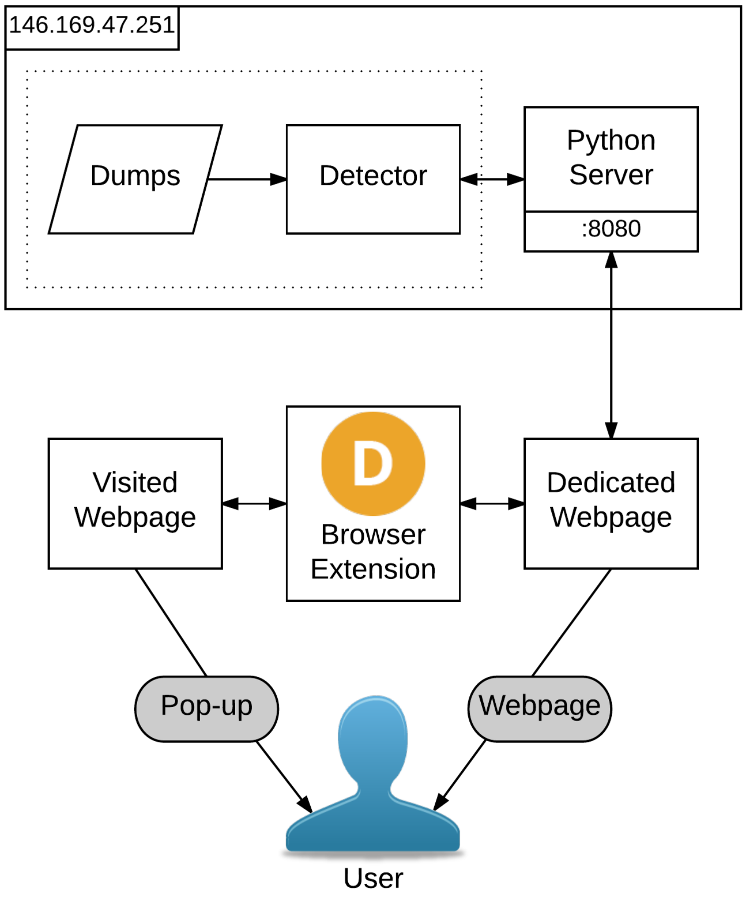


Figure 4.5: Architecture of the online detector

Figure 4.5 shows the architecture of the detector and its interactions with an user.

First the detector, that is to say the program that predicts whether a page is malicious or not, and the Dumps folder containing the classifiers, are located in a Virtual Machine (actually in Imperial College cloud, at the IP address 146.169.47.251). The detector entity calls the function `predict()`, which given a classifier found in Dumps and an URL, returns an indication on whether the URL is malicious or benign. Note that the function `predict()` uses PhantomJS as web driver in Selenium, instead of Firefox. Indeed, the function is called in a Virtual Machine which does not have any graphical interface and therefore cannot open a "real" Firefox Browser.

We also run a simple Python server on port 8080 which links the detector to the outside world. When a user visits the URL `http://146.169.47.251:8080/prediction?url=<URL_to_predict>`, the server calls the Detector entity and asks for the result of its prediction of `URL_to_predict`. The Detector entity then returns the results to the server which displays in the user's browser either 'benign' or 'malicious'.

Because it is not always convenient to visit an external webpage before visiting the wanted webpage, we created a browser extension aiming at alerting the user on whether the visited page is malicious or not. The extension runs on Firefox. Each time an user clicks on a URL, this URL is sent to our web server, which calls the detector. An alert pop-up is then displayed when the browser gets the response from the server.

4.4 Statistics

As mentioned in section 3.2.5, we predict whether the URLs that are at a certain “click depth” from the root URL are malicious or not. We want an automated way to create this tree and to visualize it. It is trivial to create such a tree recursively with Python. We then turn it into a JSON file whose form is described in Figure 4.6.

```
{  "name": /* URL name */,
   "type": /* Benign or malicious */,
   "children": [{  "name": /* URL name */,
                   "type": /* Benign or malicious */,
                   "children": ...
                },
                {  "name": /* URL name */,
                   "type": /* Benign or malicious */,
                   "children": ...
                }, ...
               ]
}
```

Figure 4.6: Tree at JSON format

An excellent solution to visualize such JSON files is the Javascript library D3 [63]. An example is shown in section 5.6.1.

Chapter 5

Experiments and results

We conducted an experiment with the total set of features. We first describe here the average performance of the detector. After explaining which features we used for the two experiments, we find different validation measures helping us to to choose the best classifier to use for prediction. From that, we discuss features importances (in this section, we compare the results with another experiment using a different set of feature), false positive and negative rates. Then, we show the results of the classifier in the real world: from the click tree to the analysis of a malicious example. We end up with a comparison of our approach with the relevant ones found in the literature.

5.1 Performance

Table 5.1 contains measures regarding data collection and global performance of the detector.

Number of benign URLs	440
Number of malicious URLs	257
Total analysed time	1h
Mean analysed time	5.2 s
Fitting time for SVM	59 ms
Fitting time for Decision Tree	6 ms
Fitting time for Random Forest	73 ms
Fitting time for AdaBoost	268 ms
Cross validation time for SVM	195 ms
Cross validation time for Decision Tree	79 ms
Cross validation time for Random Forest	203 ms
Cross validation time for AdaBoost	627 ms
Average prediction time with Random Forest	4.5 s

Table 5.1: Performance measurements

We managed to get 440 of the 500 most popular webpages in the United Kingdom. The others were either unavailable at crawing time, or exceeded the timeout we set for the experiment. 257 malicious webpages were found in the two blacklists we looked at. Obviously, the dataset is neither big nor representative enough, but constitutes a first basis for our detector.

Those URLs were analysed in one hour, which corresponds to 5.2 seconds per page. This a coherent result knowing that the program opens Firefox, goes on the webpage and extracts the features. Figure 5.1 shows the histograms of analysis times for benign and malicious URLs. We see that most of the benign pages are analysed between 4 and 10 seconds whereas the majority of malicious pages are analysed more quickly: between 2 and 6 seconds. This is due to the fact the benign pages we have collected (which are in fact the popular websites) contain more content than the malicious ones.

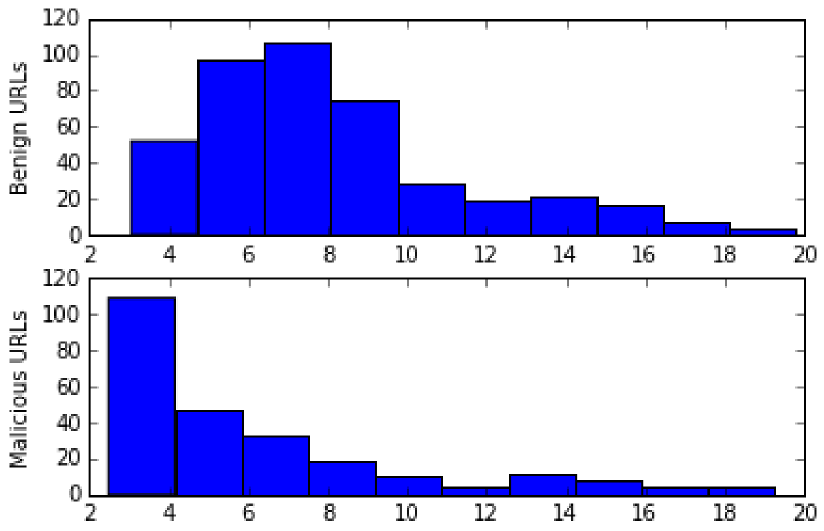


Figure 5.1: Histograms of analysis time (x-axis) for benign and malicious URLs

The analysing time can be improved by parallelising the opening of the browsers on several machines thanks to Selenium Grid [64]. In addition, instead of opening and exiting a new browser each time, a solution is to open one and only one browser for the whole experiments and for each page to analyse, create a new tab.

Fitting and cross validation times are very low ($\sim$ 1 ms) because the datasets we use are very small and the Python's `sklearn` built-in functions are very efficient and powerful. Still, we can compare the running time of each of the classifiers. Decision Tree is the quicker one because its underlying algorithm computes a tree only based on the gain of each feature on the dataset. Random Forest and SVM are ten times slower, and AdaBoost is the worse one in term of performance, with a fitting time 40 times greater than Decision Tree.

To compute the average prediction time, we chose Random Forest classifier (as discussed in section 5.3) and we applied it on the training dataset, that is to say 697 URLs. Figure 5.2 shows the histogram of prediction times.

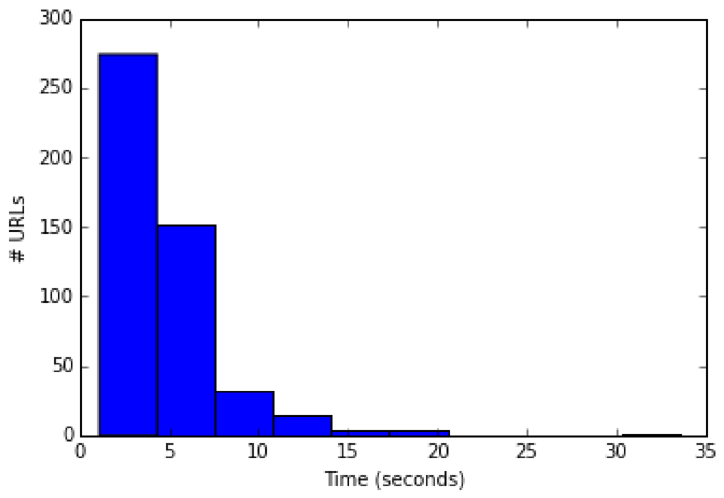


Figure 5.2: Histogram of prediction times (x-axis) for the URLs of the training dataset

The average prediction time is 4.5 seconds and we see that most of the URLs have a prediction time between 1 and 10 seconds. This is less than the average analysis time, which is understandable knowing that we use PhantomJS WebDriver (as explained in section 4.3) for the detection, a web driver which does not need to open an interface for its browser.

5.2 Features used

Table 5.2 describes the features we extracted from the webpages for two experiments (E1 and E2). The first one consists in classifying webpages with both HTML and Javascript features. The second one consists in classifying data with only Javascript features, it is performed in section 5.4.

Feature	**E1**	**E2**	**Description**
html_letter_count	x		Number of letters in the whole HTML
html_word_count	x		Number of words in the whole HTML
html_line_count	x		Number of lines of the whole HTML
js_scripts_count	x	x	Number of Javascript snippets and/or files in the page
js_min_letter_count	x	x	Number of letters of the script containing the less letters among all of the extracted scripts
js_max_letter_count	x	x	Number of letters of the script containing the maximum letters among all of the extracted scripts
js_total_letter_count	x	x	Sum of letters of all the extracted scripts
js_min_word_count	x	x	Number of lines of the script containing the less lines among all of the extracted scripts
js_max_word_count	x	x	Number of words of the script containing the maximum words among all of the extracted scripts
js_total_word_count	x	x	Sum of words of all the extracted scripts
js_min_line_count	x	x	Number of lines of the script containing the less lines among all of the extracted scripts
js_max_line_count	x	x	Number of lines of the script containing the maximum lines among all of the extracted scripts
js_total_line_count	x	x	Number of words per line for all the extracted scripts
js_min_letter_by_word	x	x	Number of letters per word of the script containing the less words among all of the extracted scripts
js_max_letter_by_word	x	x	Number of letters per word of the script containing the maximum words among all of the extracted scripts
js_total_letter_by_word	x	x	Number of letters per word for all the extracted scripts
js_min_word_by_line	x	x	Number of words per line of the script containing the less words among all of the extracted scripts
js_max_word_by_line	x	x	Number of words per line of the script containing the maximum words among all of the extracted scripts
js_total_word_by_line	x	x	Number of words per line for all the extracted scripts
js_total_eval_count	x	x	Count of the `eval` keyword in all the extracted scripts
js_total_escape_count	x	x	Count of the `escape` keyword in all the extracted scripts
js_total_unescape_count	x	x	Count of the `unescape` keyword in all the scripts
js_total_whitespace_count	x	x	Count of the whitespace characters in all the scripts

Table 5.2: Description of the features used by our detector

There are only static features coming from the HTML and Javascript. We compute the letter, word and line counts for the HTML and also for each of the Javascript files and snippets collected for one page. From the counts of all Javascripts, the minimum, maximum and sum of each of the metrics among the Javascript files and snippets are actual features. The number of Javascripts files and snippets from each page is also a feature as well as the average number of letters per word and words per line. Also, according to the best practices found in the literature, we compute the number of times certain keywords are found in the Javascript. Those keywords are `eval`, `escape` and `unescape`, which are widely used to obfuscate code. Finally, we also count the number of whitespace characters contained in the Javascript. Although we should definitely extract dynamic features, the static ones give interesting results and form a good basis for the detector.

E1 is the principal experiment and all the following sections refer to it. Only section 5.4 compares results found for E1 and E2.

5.3 Validation measures

After the cross validation is performed in our dataset, we compute validation metrics for the four classifiers (SVM, Decision Tree, Random Forest, AdaBoost). Results are shown in Table 5.3.

	SVM	Decision Tree	Random Forest	AdaBoost
TP	63%	51%	58%	56%
FP	26%	9%	12%	9%
FN	**0%**	12%	5%	7%
TN	10%	27%	25%	28%
Accuracy	74%	79%	**83%**	**83%**
Precision	71%	84%	83%	**86%**
Recall	1%	81%	**92%**	88%
F_2 score	**92%**	82%	90%	88%
NPP	1%	70%	**83%**	79%
Specificity	28%	74%	68%	**75%**

Table 5.3: Validation measures

We aim at using the classifier with the best overall validation performance. An important condition is to have a low rate of false negative but we do not specially focus on false positive. Indeed, classifying a malicious item as benign is much more harmful than classifying a benign item as malicious. The false positive rate must be reduced as much as possible but not to false negative rate's disadvantage.

Although SVM shows a null false negative rate, its performance is very low compared to the three other classifiers. Note that we used the basic Python's implementation of SVM classifier. The results can undoubtedly improve with a customized algorithm. In the current state, we do not select this classifier as our final one. We do not select Decision Tree neither as each of the metrics measured is lower than the other classifiers.

The choice has to be made between Random Forest and AdaBoost, which show quite similar measures: 83% accuracy, around 85% precision, around 90% recall and F_2 score, a NPP of approximately 80% and a specificity varying between 68% and 75 % for respectively Random Forest and AdaBoost. The determining factor is, as mentioned above, the false negative rate and as Random Forest has a lower one, we select this classifier for the rest of our project.

With such a low amount of items in our dataset, the results are not significant in absolute terms. However they give an idea of which classifiers are more likely to be performant than others.

5.4 Best features

Among the different extracted features, it is interesting to see which of them have contributed the more to classify the data. There is no built-in function or algorithm allowing to determine which feature has the more importance for SVM and AdaBoost.

One first and naive approach is to test features individually, feed the classifiers with the value of this single feature for each of the items, and look at the validation metrics to determine which of the features have the most impact on the data. This solution is not efficient at all.

Another solution is to do it via Random Forest and Decision Trees and since the results found for the Random Forest classifier in section 5.3 are the best, this method is relevant. Let us recall that Random Forest apply Decision Tree algorithm several times and average the results. The importance of a feature in Random Forest is inherent of the algorithm (and in particular the Decision Tree algorithm) and is represented by its gain (averaged gain in the case of Random Forest). The Decision Tree algorithm consists in creating a tree according to the gains of all the features and thanks to Python's package `sklearn` [59], a list containing the gain of all features is stored into the classifier. Random Forest classifier contains then a list of the averaged gains. It is easy and very convenient to get this list. The importance (derived from the gain) of each of the features used in E1 (see section 5.2) is found in Table 5.4.

Feature	**Importance**
html_letter_count	**46.4%**
html_word_count	**9.0%**
html_line_count	2.9%
js_scripts_count	2.2%
js_min_letter_count	1.0%
js_max_letter_count	3.0%
js_total_letter_count	0.0%
js_min_word_count	7.4%
js_max_word_count	1.7%
js_total_word_count	1.4%
js_min_line_count	0.5%
js_max_line_count	4.1%
js_total_line_count	1.0%
js_min_letter_by_word	2.3%
js_max_letter_by_word	0.9%
js_total_letter_by_word	1.2%
js_min_word_by_line	2.0%
js_max_word_by_line	**8.5%**
js_total_word_by_line	1.9%
js_total_eval_count	0.6%
js_total_escape_count	0.9%
js_total_unescape_count	0.0%
js_total_whitespace_count	1.3%

Table 5.4: Feature importances for the whole set of features (E1)

According to the table, the feature that has the biggest impact on the classification is the HTML letter count. The second one is the word count in the HTML. Knowing that the benign dataset contains popular websites that generally host a large amount of content, and given that malicious websites crawled are pages with much less content, this is not surprising. However, for a real world detection, this is problematic. Indeed, small websites are more likely to be flagged malicious. 100% of the the personal websites (all are light websites with not much content) of Imperial friends that we have tested are flagged as malicious ones while they are obviously benign. This issue can be fixed by training the classifier on a more representative dataset containing all sorts of webpages: from very small to huge.

The number of words per line in the biggest Javascript file or snippet is also determining in the classification. Attackers often use obfuscation to hide their malicious code. Adding noise to the code is one of the obfuscation techniques and this can be an explanation here.

According to the literature, the number of times keywords like `eval`, `unescape` or `escape` appear in the code should have a much bigger weight in the classification. They do not exceed 2% of importance here. The explanation lies again in the data collection. Malicious pages are not all malicious in the blacklists and the dataset is not huge and representative enough.

In order to reduce the bias induced by the size of the webpages, we decided to run another experiment (E2) in which we classify the data only with Javascript features (see section 5.2). The new importances are shown in Table 5.5.

Feature	**Importance**
js_scripts_count	5.2%
js_min_letter_count	1.9%
js_max_letter_count	6.2%
js_total_letter_count	4.2%
js_min_word_count	8.1%
js_max_word_count	0.0%
js_total_word_count	3.0%
js_min_line_count	0.3%
js_max_line_count	1.1%
js_total_line_count	0.7%
js_min_letter_by_word	1.7%
js_max_letter_by_word	3.2%
js_total_letter_by_word	2.0%
js_min_word_by_line	1.6%
js_max_word_by_line	**18.0%**
js_total_word_by_line	3.2%
js_total_eval_count	1.1%
js_total_escape_count	**30.0%**
js_total_unescape_count	3.1%
js_total_whitespace_count	2.3%

Table 5.5: Feature importances for only Javascript features (E2)

Unsurprisingly, words by line is the biggest Javascript plays a big part of the classification. This corroborates the results found in the first experiment (E1) seen above. Furthermore, we notice that the count `escape` keyword is determining in the classification (30% importance). This is a classical obfuscation feature whose effect was somehow hidden by the HTML features in experiment E1. Still, `eval` and `unescape` keywords should have a higher gain. This is both due to the the size of the datasets and to advanced obfuscation techniques used by attackers to hide the calls to these functions (see the example of malicious page in section 5.6.2)

5.5 False positive and false negative

As explained above, we focus on reducing the number of false negative items rather than false positive ones. Indeed, if an user visits a malicious webpage that our detector detects as benign, this can be harmful and trigger unexpected and undetected behaviours. This is much more problematic than false positives in that if a user visits a webpage flagged as malicious, if it is in fact benign, it will not have a negative impact and trigger malicious behaviours. The ideal case would obviously be to neither have false positive nor false negative items...

Table 5.3 of indicates that Random Forest classifier (the algorithm we chose for our detector) has a false positive rate of 12% and a false negative rate of 5%.

False positive examples are pages that “look like” the pages stored into the malicious dataset and do not “look like” the pages stored into the benign dataset. As explained in section 5.4, those false positive items are mostly pages containing a few letters and words like those stored into the malicious dataset. It is interesting to see that the malicious websites from the blacklists we crawled are often rapidly taken down by their administrator, and sometimes show an error message alerting that the page has been hacked or is in maintenance. These messages contain a low amount of letters and words, which corroborates the fact that the number of letters in the HTML has a great impact on the classification. One direction is then to create a filter that analyses the blacklisted pages and discards the taken down ones. For this project we only created a filter that discards the pages that sends an HTTP error code.

False negative items are pages that look benign but are not. We manually inspected all the false negative examples detected and we found nothing malicious in them.

5.6 Real-world deployment

5.6.1 Tree

As explained in section 3.2.5, we used D3 [63] to compute and draw a prediction tree as shown in Figure 5.3. Each node is an URL. The central one is the root: the website on which we first click. This node is linked to nine others who represent the URLs accessible from the root page. Each of those nodes are linked to their children. In our case, the tree has a click depth of 2.

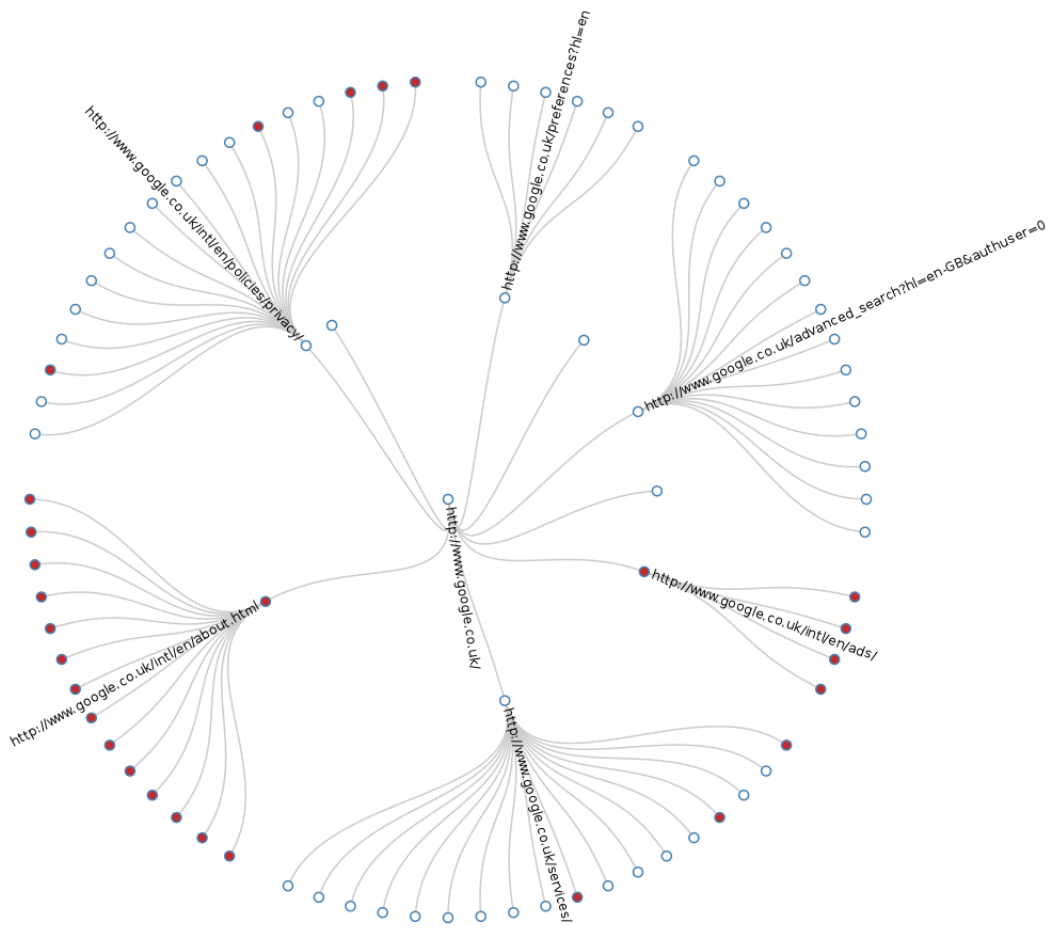

http://www.google.co.uk/
http://www.google.co.uk/intl/en/policies/privacy/
http://www.google.co.uk/preferences?hl=en
http://www.google.co.uk/advanced_search?hl=en-GB&authuser=0
http://www.google.co.uk/intl/en/ads/
http://www.google.co.uk/services/
http://www.google.co.uk/intl/en/about.html

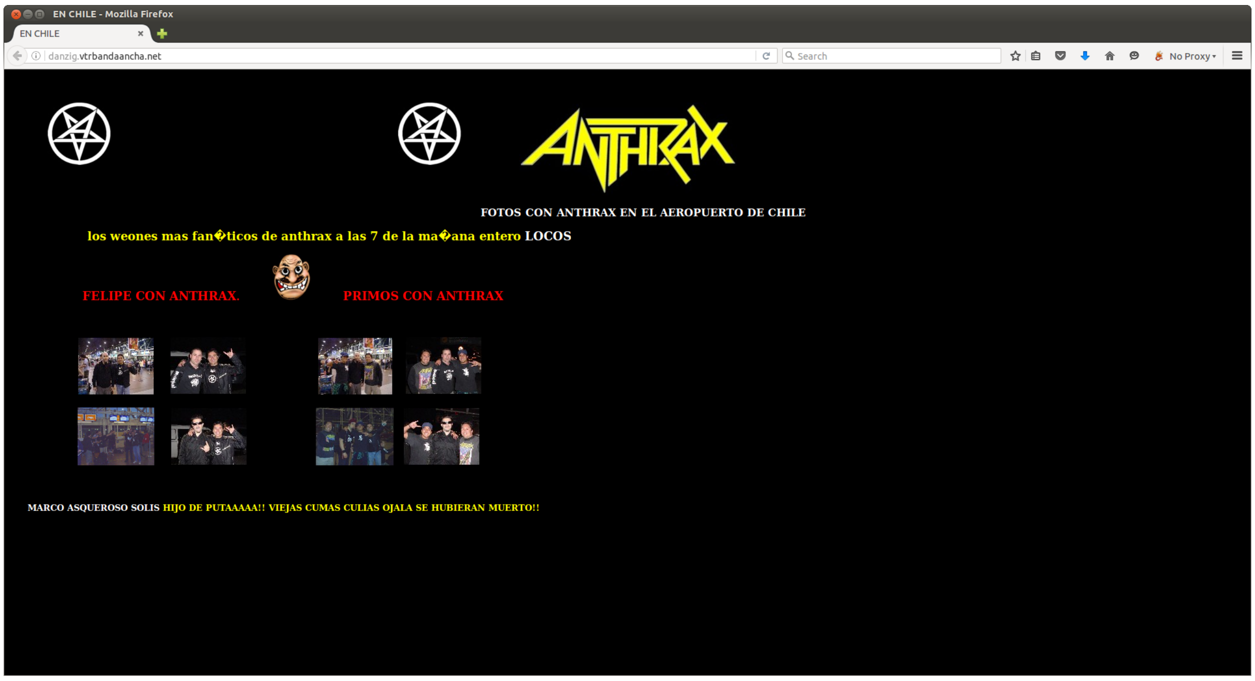
Figure 5.3: Tree of predictions for a click depth of 2 from `http://www.google.co.uk/`

A node is red if it is flagged as malicious by our detector, and white when it is flagged as benign. Here the root node corresponds to the url `http://www.google.co.uk/`. The URLs accessible from it are all from the same domain. Our detector flagged two of those URLs as malicious. Actually, there are benign but considered as false positive items for the same reasons explained in section 5.4.

The links extracted from those two malicious webpages are also flagged as malicious. This is because they are links from the same domain and derived from the same initial page. So the features extracted from them are similar to those extracted from the first malicious links. Among the 52 other links of click-depth 2, 8 are labeled malicious, and after manual inspection they all are benign, so it corresponds to a 15% false positive rate, which is coherent with the validation measures from section 5.3.

5.6.2 Example of a malicious webpage

It is interesting to look at real examples of malicious pages and scripts to make our techniques evolve and be more efficient. We see here the example of `http://danzig.vtrbandaancha.net`, labeled as malicious by our detector and added to the Malware Domains blacklist [45] on September 2016. As shown in Figure 5.4, the webpage does not look nice and warm at first sight...



FOTOS CON ANTHRAX EN EL AEROPUERTO DE CHILE

los weones mas fan�ticos de anthrax a las 7 de la ma�ana entero LOCOS

FELIPE CON ANTHRAX. PRIMOS CON ANTHRAX

MARCO ASQUEROSO SOLIS HIJO DE PUTAAAAA!! VIEJAS CUMAS CULIAS OJALA SE HUBIERAN MUERTO!!

Figure 5.4: Screenshot of a malicious website

Figure 5.5 shows the structure of the page source.

```
1  <? Malicious PHP  ?>
2  <iframe style="visibility:hidden;display:none" src="host.php" ></i↩
     frame><html>
3  <head>
4  <!-- Head -->
5  </head>
6
7  <body bgcolor="#000000"><script> Malicious Javascript </script>
8  <!-- Body -->
9  </body>
10
11 </html>
```

Figure 5.5: Source structure of a malicious sample

Three elements catch the eye: the presence of a PHP snippet at the very beginning of the source code (line 1), the creation of an `iframe` calling the PHP file `host.php` and the big one-line script at line 7. Unfortunately, we cannot access `host.php`. We assume that the script has quickly been taken down. Let us first focus on the PHP snippet, given in Figure 5.6.

Line 1 turns off error reporting. That is to say, if a script triggers an error in one's browser, no information will be displayed. It is a way to hide the possible errors that can occur, more particularly in `host.php`. The aim of the second part of the code (lines 2-5) is unclear. The script tries to open the file `host.txt` on the server. If it succeeds, it writes the hostname of the server (`HTTP_HOST`) in this file. We managed to open the file and the hostname was actually written in

```php
error_reporting(0);
$mhfp = fopen("host.txt", "w");
if ($mhfp) {
  fwrite($mhfp, $_SERVER["HTTP_HOST"]);
  fclose($mhfp);
}
```

Figure 5.6: PHP snippet of a malicious sample

it. The author of the code maybe wanted to have an indication on whether an user visited the webpage and then felt into the trap.

The most interesting part of this example is the Javascript. Figure 5.7 shows the unpacked version of the big one-line script found in the source code (line 7 of Figure 5.5).

```javascript
b=new function(){return 2;};
if(!+b)
  String.prototype.vqwfbeweb='h'+'arC';

for(i in $='b4h3tbn34')
  if(i=='vqwfbeweb')
    m=$[i];

try{
  new Object().wehweh();}
catch(q){
  ss="";}

try{
  window['e'+'v'+'al']('asdas')}
catch(q){
  s=String["fr"+"omC"+m+"od"+'e'];}

d=new Date();
d2=new Date(d.valueOf()-2);

Object.prototype.asd='e';
if({}.asd==='e')
  a=document["c"+"r"+"e"+"a"+"t"+"e"+"T"+"e"+"x"+"t"+"N"+"o"+"d"+"e"]↩
      ('321');

if(a.data==321)
  h=-1*(d-d2);

n=[-h+7,-h+7,-h+103,-h+100,-h+30, ... ,-h+57,-h+7,-h+7,-h+123];

for(i=0;i<n.length;i++)
  ss+=s(eval("n"+"[i"+"]"));

if(!+b)eval(ss);
```

Figure 5.7: Malicious Javascript of a malicious sample

The script is a good example of obfuscation and evasion. Let us decrypt the methods the author used to hide its malicious content.

On line 1, a function `b` is instantiated. Line 2 tests whether the function has correctly been instantiated. In that case, `+b` returns `NaN`, so `!+b` is satisfied and the code located at line 3 can be run. This is the first instance of evasion: if a browser does not let a script create a function, then the code at line 3 is not run. This code consists in modifying the Javascript String prototype by creating a new method `vqwfbeweb` which, when applied to a string, turns it into the string `"harC"`. Note that splitting `"harC"` into `'h'+'arc'` is a well-known obfuscation technique aiming at evading static detectors (see section 2.2.2).

Lines 5-7 is a loop such as: for every String methods of the random string `"b4h3tbn34"` (which looks like leet code by the way), if the method is `vqwfbeweb`, then the variable `m` takes the value of the applied method, that is to say: `"harC"`. With the if condition, the script checks whether the prototype has actually been modified. Therefore, it is again an evasion measure: a super-browser that provides the user to modify prototypes will not be targeted by the malicious script.

Line 10 tries to instantiate a new Object named `wehweh`. Except in the unlikely case where `wehweh` is already instantiated in the user's browser, this line will always fail and trigger the code in the catch, which is the instantiation of the variable `ss` by `""`. Some detectors only scan the "try" expression of try-catch. The script here wants therefore to hide the instantiation of the `ss` variable that plays a great part in the script.

The next block (lines 14-17) is similar to the previous one: evaluating the function `asdas` always fails and trigger the catch code, which instantiates the variable `s` to the String method `String["fr"+"omC"+m+"od"+'e']`, which corresponds, according to the signification of `m` seen above, to: `String["fromCharCode"]`.

Lines 19 and 20 instantiate two dates which values differ of two. Note that substracting `d2` to `d` returns 2. This will have its importance at the end of the script.

Line 22 modifies the Object prototype by creating a new method `asd` return the character `'e'`. Line 23 tries to call this method from a random object (here {}) and checks if it returns the right result. This is a test to know whether the browser accepts that the script modifies the prototype of Object. If it does, the variable `a` is instantiated with the string `'321'` thanks to the function `createTextNode` (the splitted version) of `document`.

Line 26 tests whether the previous block has succeeded. If yes, the variable `h` is instantiated with `-1*(d-d2)`, which corresponds to -2 as `d-d2 = 2`.

On line 29, a list `n` of length 799 is created and each of its cell is a number of the form `-h + x` where `x` is an integer. We can expect these numbers are ASCII characters. As we can see, numbers are splitted, they are not written basically. Indeed, this is another well-spreaded obfuscation techniques, that tricks algorithms that detect scripts using the decimal ASCII representation of a character (see section 2.2.2).

Given that each cell of `n` is an ASCII character, `s` is the String method `fromCharCode` and `ss` is an empty string, it is easy to see that lines 31 and 32 append in `ss` all the deobfuscated character of `n`. Line 34 simply evaluates the string `ss`.

Before looking at `ss`, it is important to note that each time the function `eval` is mentioned, it is sticked to other characters. The majority of detectors parse the scripts and look for the count of `eval` words. The result here would be 0 thanks to this obfuscation technique.

Let us get back to the `ss` string evaluated in line 34... it is in fact another Javascript code, and, as shown in Figure 5.8, it is this time almost entirely deobfuscated!

```
if (document.getElementsByTagName('body')[0]){
  iframer();
}else{
  document.write("<iframe src='http://%71%65%72%66%68%67%6B%61%64%68%73%66%75%6B%68%65%72%74%67%72%70%6F%74%67%6A%70%6F%69%64%66%67%2E%63%65%2E%6D%73/main.php?page=b5a87c34230be775' width='10' height='10' style='visibility:hidden;position:absolute;left:0;top:0;'></iframe>");
}

function iframer(){
  var f = document.createElement('iframe');
  f.setAttribute('src','http://%71%65%72%66%68%67%6B%61%64%68%73%66%75%6B%68%65%72%74%67%72%70%6F%74%67%6A%70%6F%69%64%66%67%2E%63%65%2E%6D%73/main.php?page=b5a87c34230be775');
  f.style.visibility='hidden';
  f.style.position='absolute';
  f.style.left='0';
  f.style.top='0';
  f.setAttribute('width','10');
  f.setAttribute('height','10');
  document.getElementsByTagName('body')[0].appendChild(f);
}
```

Figure 5.8: Deobfuscated malicious Javascript of a malicious sample

The deobfuscated script simply creates an `iframe` in the body of the page when it has one, or out of it else. This `iframe` points toward an obfuscated website, which, when decoded refers to: `http://qerfhgkadhsfukhertgrpotgjpoidfg.ce.ms/main.php?page=b5a87c34230be775`.

Unfortunately, the ultimate website was not available at the time of the analysis. It may have been taken down. According to the blacklist where we get the first malicious URL, this one classifies the URL as a gateway to Locky exploit kit. Therefore, we can expect that the final URL that we found would have triggered a drive-by-download attack, installing the Locky exploit kit, before being taken down.

A quick search online showed that this malicious script is quite popular [65] among the attacker community.

5.7 Comparison with other approaches

It is interesting to compare the results of our detector with existing approaches, in particular performance from the several articles surveyed in the section 2. However, this comparison is not quite relevant since the datasets of the other methods were much larger that our detector's. With this point in mind, let us compare the detector with the other ones through Table 5.6.

	Our detector	[16]	[18]	[30]	[5]	[25]	[9]
TP	58%	-	-	-	-	-	-
FP	12%	-	0.002%	0.21%	~0%	9.88%	0.51%
FN	5%	-	-	-	0.2%	0.77%	-
TN	25%	-	-	-	-	-	-
Accuracy	83%	-	-	96%	-	-	98.5%
Precision	83%	88.2%	-	-	-	-	-
Recall	92%	78.7%	90.2%	85%	99.8%	-	94%
F_2 score	90%	80.6%	-	-	-	-	-
NPP	83%	99.1%	-	-	-	-	99.4%
Specificity	68%	-	-	-	-	-	-

Table 5.6: Comparison between the validation metrics of our detector and approaches found in the literature

False positive rate is very high compare to the other approaches. It is only similar to Prophiler's [25] which is aimed at filtering the pages rather than classifying them alone. Therefore, as things stand, our detector cannot even pretend to be a filter given that its false negative rate is too high (5%) compared to other methods (0.77% for Prophiler and 0.2% for JSAND [5]).

With respectively 92%, the recall – or true positive rate – of our detector is higher than the one found by Likarish et al. [16] (78.7%), Cujo [18] (90.2%) and Houa et al. [30] (85%). It means that pages labeled as malicious are more likely to be malicious with our detector than with the previously cited approaches. However the recall is still worse than JSAND's [5] (99.8%) and Nunan et al. [9] (94%). The 83% precision and 90% F_2 score of the detector are also good results compared to the respectively 88.2% and 80.6% found by Likarish et al. [16].

However our method shows worse accuracy and NPP than others. While Houa et al. [30] and Nunan et al. [9] find accuracies of 96% and 98.5%, our detector only performs 83%. It means that the detector is globally less reliable in labeling both malicious and benign pages. Our NPP – or true negative rate – (83%) is also lower than Likarish et al. (99.1%) and Nunan et al. (99.4%).

Chapter 6

Conclusions and future work

This project is an exploration of the world of the detection of malicious behaviours on the Internet using machine learning. Although, as things stand, it will not provoke a revolution or make the world a better place, it allows us to have a broader view on the intersection of two enthralling fields: security and machine learning.

The detector we have created is not perfect yet but the skeleton is a solid basis on which we can build interesting programs in the future. There are several axis for improvements:

1. We need to enlarge our benign and malicious datasets that is too small (a total of 700 pages). A simple way would be to get thousands of pages from the charged Alexa API, and, after sanitizing, label them as benign. For the malicious dataset, a direction is to make our "blacklist crawling" script running continuously on a server and accessing various blacklists or other sources on the internet. As soon as a new URL is fetched by this automated script, our detector would save the HTML and Javascript and extract the relevant features. Furthermore, malicious URLs need to be inspected and sanitized in order to remove false positives.

2. We currently extract 23 features from the pages. It would be interesting to do a advanced study on the choice of features we use, test several sets of the features and select the most relevant ones. One can also try to apply the q-grams method which does not infer any meaning into the features. The comparison between the q-grams approach and the meaningful features has never been done for a same approach. Also, adding dynamic features is important in that they can detect obfuscated malicious behaviours.

3. A good axis of improvement is to compare the results when using different web drivers to open a page and extract features from it. Often, malicious scripts have fingerprinting mechanisms allowing them to trigger malicious behaviours only if they are run in a "regular" browser. It would be therefore interesting to compare the impact of a browser over an other on the results. Our detector analyses the pages with a Firefox browser similar than the one regular users use. But in order to analyse a page, the detector opens it on Firefox, extracts the features and closes it. It takes an average of 4.5 seconds per page, which does not correspond to a "regular" behaviour and malicious scripts could fingerprint it and evade the detector! We must then think to anti-evasion approaches to remain ahead of bad people.

4. Our approach compares 4 basic classifiers. We can use more classifiers and even customize them when needed.

5. A good approach is to create demo websites infected with different Exploit Kits (e.g. Angler or Nuclear) found in the wild and feed our malicious database. This could be used either for training our classifiers or for evaluating the detector's performance.

Bibliography

[1] Internet World Stats. `http://www.internetworldstats.com/stats.htm`.

[2] Symantec Internet Security Threat Report 2015. `https://www4.symantec.com/mktginfo/whitepaper/ISTR/21347932_GA-internet-security-threat-report-volume-20-2015-social_v2.pdf`.

[3] ILOVEYOU Virus. `http://searchsecurity.techtarget.com/definition/ILOVEYOU-virus`.

[4] Kapersky Report. `http://www.kaspersky.com/internet-security-center/threats/types-of-malware`.

[5] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and Analysis of Drive-by-download Attacks and Malicious JavaScript Code. *Proceedings of the 19th international conference on World wide web*, 2010.

[6] Common Vulnerabilities and Exposures. `https://cve.mitre.org/`.

[7] Panopticlick. `https://panopticlick.eff.org/`.

[8] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: a Client-side Solution for Mitigating Cross-Site Scripting Attacks. *Proceedings of the 2006 ACM symposium on Applied computing*, 2006.

[9] Angelo Eduardo Nunan, Eduardo Souto, Eulanda M. dos Santos, and Eduardo Feitosa. Automatic Classification of Cross-Site Scripting in Web Pages Using Document-based and URL-based Features. *IEEE Symposium on Computers and Communications (ISCC)*, July 2012.

[10] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. *S&P'06 IEEE Symposium on Security and Privacy*, 2006.

[11] Roberto Perdisci, Wenke Lee, and Nick Feamstera. Behavioral Clustering of HTTP-Based Malware and Signature Generation Using Malicious Network Traces. *USENIX Symposium on Networked Systems Design and Implementation*, 2010.

[12] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. *IEEE Symposium on Security and Privacy (S&P'05)*, 2005.

[13] Snort. `https://www.snort.org/`.

[14] W. Xu and F. Zhang and S. Zhu. The Power Of Obfuscation Techniques In Malicious JavaScript Code: A Measurement Study. *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on*, pages 9–16, Oct 2012.

[15] The International Obfuscated C Code Contest Home Page. `http://www.ioccc.org/`.

[16] Peter Likarish, Eunjin (EJ) Jung, and Insoon Jo. Obfuscated Malicious Javascript Detection using Classification Techniques. *4th International Conference on Malicious and Unwanted Software (MALWARE)*, October 2009.

[17] HTMLUnit. `http://htmlunit.sourceforge.net/`.

[18] Konrad Rieck, Tammo Krueger, and Andreas Dewald. Cujo: Efficient Detection and Prevention of Drive-by-download Attacks. *Proceedings of the 26th Annual Computer Security Applications Conference*, 2010.

[19] Stephen C. Johnson. Yacc: Yet Another Compiler-Compiler. `http://dinosaur.compilertools.net/yacc/`.

[20] Andreas Dewald, Thorsten Holz, and Felix C. Freiling. ADSandbox: Sandboxing JavaScript to fight Malicious Websites. *SAC '10 Proceedings of the 2010 ACM Symposium on Applied Computing*, 2010.

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

[22] Alexandros Kapravelos, Yan Shoshitaishvili, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Revolver: An Automated Approach to the Detection of Evasive Web-based Malware. *USENIX Security*, 2013.

[23] Wepawet. `http://wepawet.cs.ucsb.edu`.

[24] Kristof Schütt, Marius Kloft, Alexander Bikadorov, and Konrad Rieck. Early Detection of Malicious Behavior in JavaScript Code. *Proceedings of the 5th ACM workshop on Security and artificial intelligence*, 2012.

[25] Davide Canali, Marco Cova, Giovanni Vigna, and Christopher Kruegel. Prophiler: a Fast Filter for the Large-scale Detection of Malicious Web Pages. *Proceedings of the 20th international conference on World wide web*, 2011.

[26] Alexa Top Sites. `http://www.alexa.com/topsites`.

[27] Google Safe Browsing API. `https://developers.google.com/safe-browsing/`.

[28] Dmoz. `http://www.dmoz.org`.

[29] ClueWeb09. `http://www.lemurproject.org`.

[30] Yung-Tsung Houa, Yimeng Changb, Tsuhan Chenb, Chi-Sung Laihc, and Chia-Mei Chena. Malicious Web Content Detection by Machine Learning. *Expert Systems with Applications*, 2010.

[31] Malekal. `http://www.malekal.com`.

[32] MalwareURL. `http://www.malwareurl.com`.

[33] MWCollect. `https://alliance.mwcollect.org/`.

[34] Malfease. `https://malfease.oarci.net/`.

[35] Javascript Compressor. `http://dean.edwards.name/packer`.

[36] StopBadWare. `https://www.stopbadware.org/`.

[37] Spamcop. `https://www.spamcop.net/`.

[38] MalwareDomainList Forum. `http://malwaredomainlist.com`.

[39] Milw0rm Forum. `http://milw0rm.com`.

[40] XXSed Database. `http://www.xssed.com`.

[41] Rich Caruana and Alexandru Niculescu-Mizil. An Empirical Comparison of Supervised Learning Algorithms. *ICML '06 Proceedings of the 23rd international conference on Machine learning*, 2006.

[42] Lecture on Boosted Decision Trees by Christian Autermann, University of Hamburg. `http://wwwiexp.desy.de/users/auterman/talks/20070706_hh_bdt.pdf`.

[43] Yoav Freund and Llew Mason. The Alternating Decision Tree Algorithm. *ICML '99 Proceedings of the Sixteenth International Conference on Machine Learning*, 1999.

[44] RIPPER description by William Cohen. `http://www.csee.usf.edu/~hall/dm/ripper.pdf`.

[45] MalwareDomains. `http://www.malwaredomains.com`.

[46] ClamAV Antivirus. `https://www.clamav.net/`.

[47] Jose Nazario. PhoneyC: A Virtual Client Honeypot. *LEET'09: Large Scale Exploits and Emergent Threats*, 2009.

[48] Capture-HPC Honeynet Project. `https://projects.honeynet.org/capture-hpc`.

[49] AVG Antivirus. `http://www.avg.com/`.

[50] Christian Seifert, Ian Welch, and Peter Komisarczuk. Identification of Malicious Web Pages with Static Analysis. *Telecommunication Networks and Applications Conference*, 2008.

[51] Justin Ma, Lawrence K. Saul, Stefan Savage, and Geoffrey M. Voelker. Beyond Blacklists: Learning to Detect Malicious Web Sites from Suspicious URLs. *KDD '09 Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data mining*, 2009.

[52] Ben Feinstein and Daniel Peck. Caffeine Monkey: Automated Collection, Detection and Analysis of Malicious Javascript Code. *Black Hat USA*, 2007.

[53] Alexa Documentation. `https://support.alexa.com/hc/en-us/articles/200449744-How-are-Alexa-s-traffic-rankings-determined-`.

[54] Biggest Data Breaches In The Last Ten Years. `http://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/`.

[55] Selenium Web Browser Automation. `http://www.seleniumhq.org/`.

[56] `urllib2` package. `https://docs.python.org/2/library/urllib2.html`.

[57] Selenium Documentation. `http://www.seleniumhq.org/`.

[58] Selenium WebDriver Documentation. `http://docs.seleniumhq.org/docs/03_webdriver.jsp#chapter03-reference`.

[59] `sklearn` Module Documentation. `http://scikit-learn.org/stable/documentation.html`.

[60] `sklearn` Supervised Learning. `http://scikit-learn.org/stable/supervised_learning.html`.

[61] `sklearn` Support Vector Machine. `http://scikit-learn.org/stable/modules/svm.html`.

[62] MongoDB Documentation. `https://docs.mongodb.com/manual/introduction/`.

[63] Javascript Library D3. `https://d3js.org/`.

[64] Selenium Grid Documentation. http://docs.seleniumhq.org/docs/07_selenium_grid.jsp#chapter07-reference.

[65] Aw Snap. Example of Malicious Script. https://aw-snap.info/articles/js-examples.php.