Imperial College
London

Department of Computing

Malicious Web Content Detection Using Machine Learning

MASTER THESIS

Author :
Antoine VIANEY-LIAUD
`amv15@ic.ac.uk`
Imperial College London

Supervised by :
Prof. Sergio MAFFEIS

August XX, 2016

Contents

Chapter 1

Acknowledgements

Chapter 2

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 innovate 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*). 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 Professor 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 by proposing new ideas for the elaboration of a new detector of malicious web content.

We describe first what sort of malicious content is targeted, that is to say malware and popular attacks. Secondly, we depict the arm race between attackers and defenders: how defenders use static analysis to detect basic malicious content, how attackers evade the detection by obfuscating their code, how obfuscation can be defeated by dynamic analysis, how attackers found ways to evade dynamic analysis and how defenders created anti-evasion mechanisms. From that point, we stress the best practices in the literature, highlight novel contributions, and explain the methodology used to build our detector.

Chapter 3

Background

3.1 Malicious content

3.1.1 Malware

Kapersky [3] classifies the different types of malware given the "threat" it poses. Figure 3.1 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.

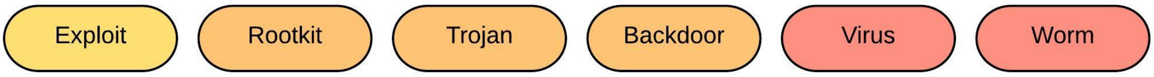


Figure 3.1: Some types of malware: yellow type poses the least threat, red ones pose greater threat [3]

One more global class of malware widely seen in the wild is drive-by-download. To Cova et al. [4], drive-by-download attacks are that successful because of:

- the abundance of existing vulnerabilities targeting browsers: 74 CVE (Common Vulnerabilities and Exposures) [5] 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 [6], an online fingerprinting tool that detects a large range of information on users' browsers.

3.1.2 Attacks

Attackers have several ways to spread malware. One of the most prolific attacks is Cross-Site Scripting (XSS) [7] [8].

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 3.2 and 3.3 are examples of respectively reflected and stored XSS.

TO-REDO

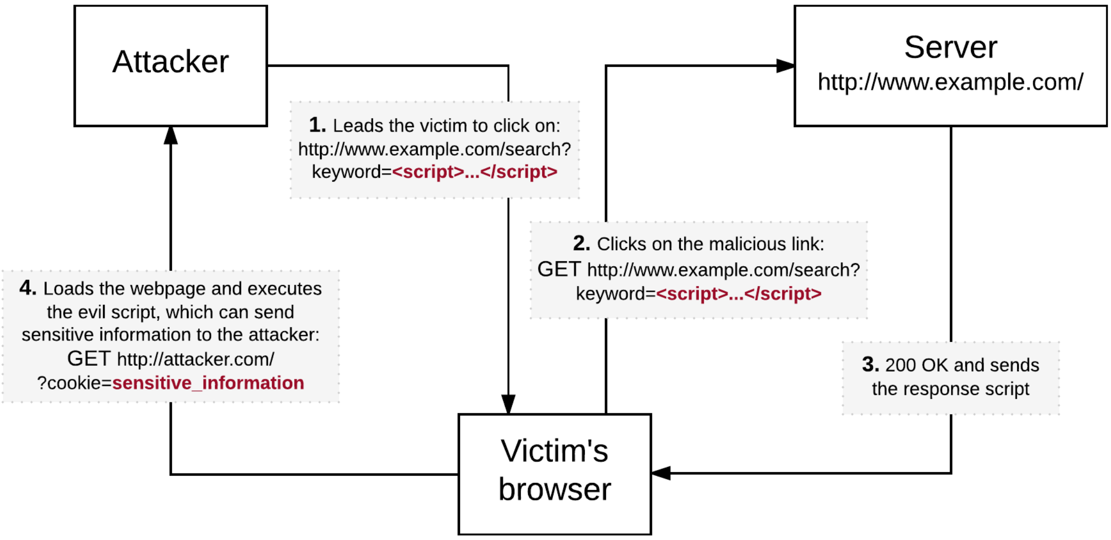

Attacker
Server
http://www.example.com/
1. Leads the victim to click on:
http://www.example.com/search?
keyword=<script>...</script>
2. Clicks on the malicious link:
GET http://www.example.com/search?
keyword=<script>...</script>
4. Loads the webpage and executes
the evil script, which can send
sensitive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

Figure 3.2: Example of reflected XSS

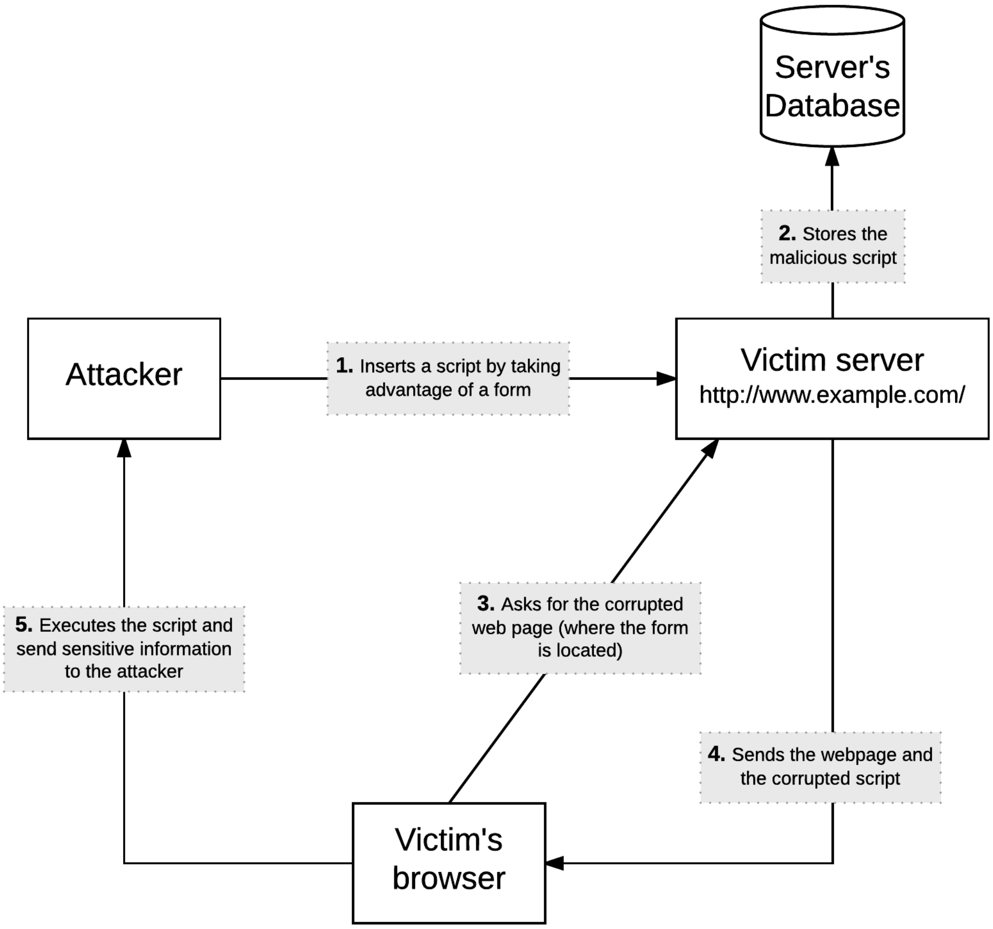


Figure 3.3: Example of stored XSS

3.2 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 3.4.

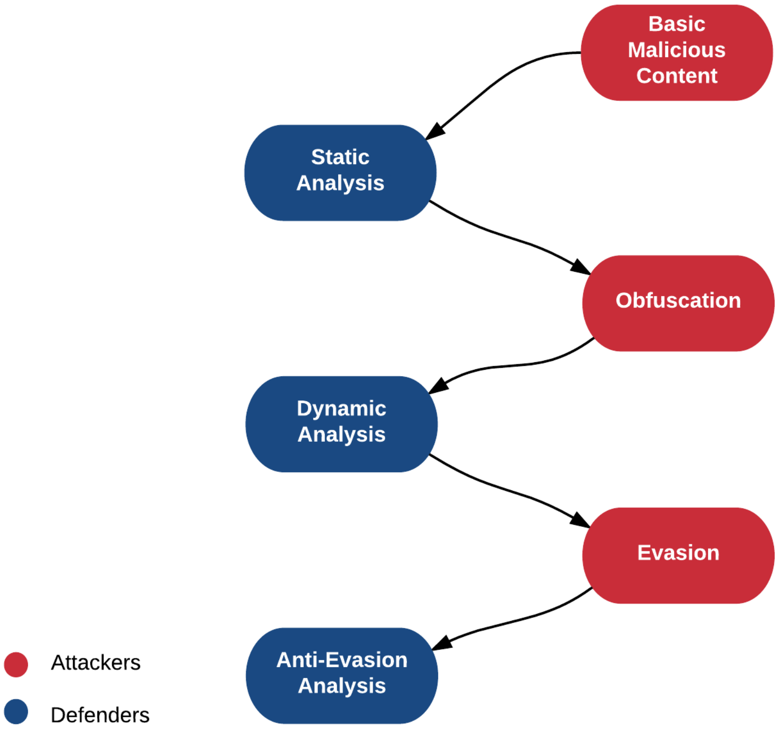


Figure 3.4: Representation of the armrace between attackers and defenders

3.2.1 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. [9] 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. [10] 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 3.5.

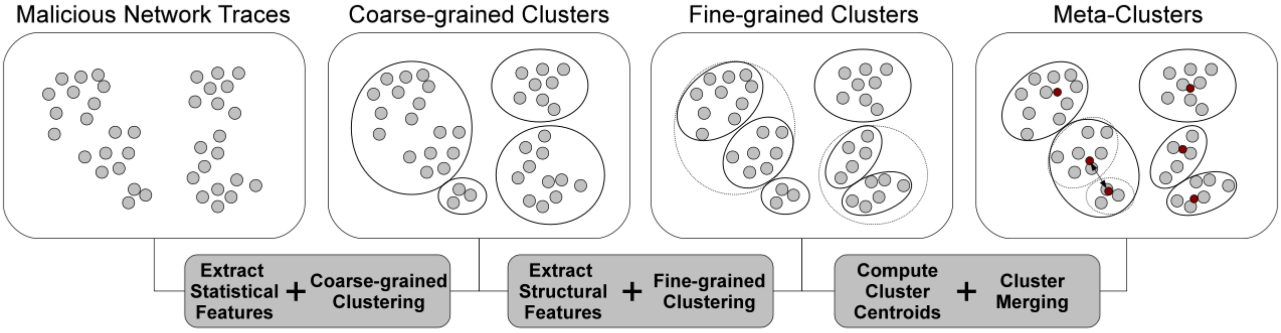

Malicious Network Traces
Coarse-grained Clusters
Fine-grained Clusters
Meta-Clusters
Extract Statistical Features + Coarse-grained Clustering
Extract Structural Features + Fine-grained Clustering
Compute Cluster Centroids + Cluster Merging

Figure 3.5: Static clustering [10]

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 3.6.

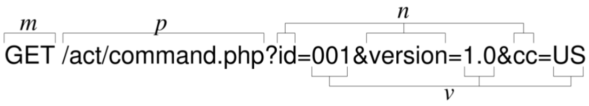


Figure 3.6: Fine-grained clustering [10]. m is the request method, p is the page name, n is the set of parameter names, v is the set of parameters values

Signatures are generated for each item thanks to the *Token-Subsequences* algorithm [11]. As shown on Figure 3.7, a signature is a list of tokens `ti` written as a regular expression of the kind: `t1.*t2.*...*tn`.

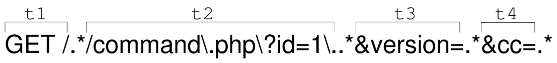


Figure 3.7: Example of signature [10]

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 [12].

Static analysis is applied to either signature-based mechanisms or code analysis. While Pixy [9] 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.

3.2.2 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. [13], there exist four types of Javascript obfuscation (cf. Table 3.1). 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 3.1: Types of obfuscation [13]

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 3.8a and 3.8b).

```
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 3.8: String splitting

Furthermore, function names can simply be replaced by new names stored in new variables (cf. Figures 3.9a and 3.9b).

Plus, 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 3.9: 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 exist obfuscation competitions awarding the most opaque pieces of code written [14].

Likarish et al. [15] suggested a way to detect Javascript obfuscation based on the use of classification techniques. They chose a large panel of features (detailed in section 3.3.2) to characterize a piece of malware and tested different classifiers (see section 3.3.3). 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. [8] 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 3.3.2 and 3.3.3. 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.

3.2.3 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 [7]: 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. [4] 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 [4]. 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 [16], 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.

3.2.4 Combination of static and dynamic analysis

Some approaches combine both static and dynamic analysis. For instance, Cujo [17], developed by Rieck et al. and illustrated in Figure 3.10, 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.

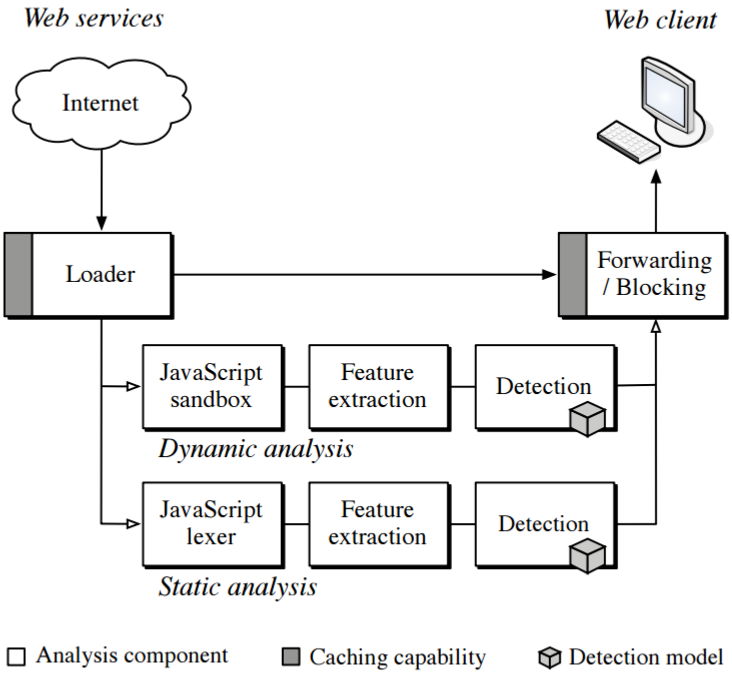


Figure 3.10: Cujo system [17]

The static analysis rests on a token extraction using a customized YACC grammar: a parser whose acronym stands for "Yet Another Compiler Compiler" [18]. 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 [19], and the Javascript interpreter SpiderMonkey [20]. 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.

3.2.5 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 [21], 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 3.11). First, the Oracle (the existing drive-by-download detection tool, which is Wepawet [22] 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).

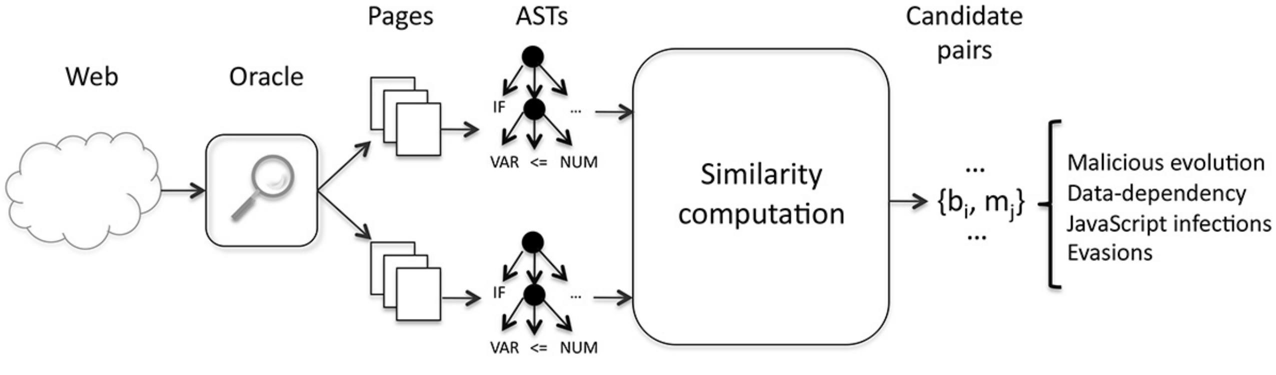


Figure 3.11: Architecture of Revolver

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 3.2.

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

Table 3.2: Candidate pairs classification (B is a benign sequence, M is a malicious sequence, * indicates a wildcard value) [21]

3.2.6 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. [23] and Rieck et al. [17] 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.

3.2.6.1 Early detection

Cujo [17] 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 3.12 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.

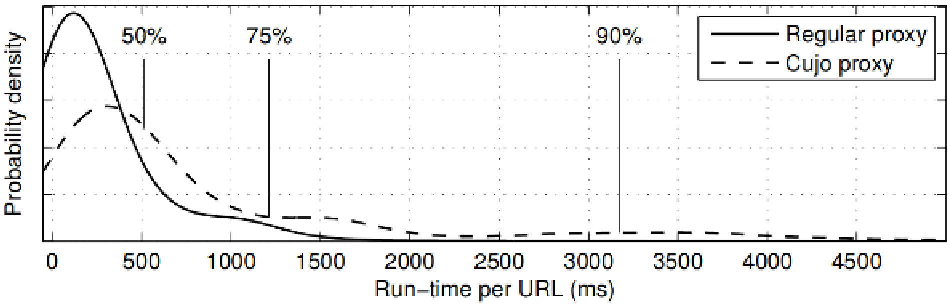


Figure 3.12: Operating run-time of Cujo and a regular web proxy on the same dataset [17]

EarlyBird [23] 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 [17] 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.

3.2.6.2 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 [4] but others aim at being used in real-time.

Canali et al. created Prophiler [24], 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.

3.3 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.

3.3.1 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 3.3 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 3.3.4).

3.3.1.1 Benign dataset

The majority of the articles surveyed [15] [17] [4] [24] [23] chose Alexa top sites [25] 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 [24] and JSAND [4]) added an extra layer of security by checking web pages crawled from Alexa with Google Safe Browsing [26] 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. [8] decided to use *Dmoz* [27] and *ClueWeb09* [28] 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. [29] were unclear about the collection of benign pages and Perdisci et al. [10] chose to sniff the supposedly secure network of a large company. Eventually, Revolver [21] 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 [22].
Note that all the authors collected more than 10,000 items except for Houa et al. [29] that used less than a thousand web pages to conduct their experiments. This seems quite low and is unlikely to be representative.

3.3.1.2 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. [15] collected only 62 malicious scripts compared to the millions of benign scripts they were able to get. For the majority of the articles [15] [17] [29] [4] [24] [23], the number of malicious items collected do not exceed 341, which seems quite low to represent all kinds of possible malicious behaviours. Prophiler [24] makes use of a little more data: 787 malicious web pages. Nunan et al. [8] and Likarish et al. [15] managed to collect respectively 15,366 and 25,720 distinct malicious items which looks more satisfying. Finally, Revolver [21] 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 [15] [17] [4] [24] [23].

Article	Malicious items		Benign items	
	Number	**Origin**	**Number**	**Origin**
Likarish et al. [15]	62 scripts	URLs blacklists [30] [31] + manual review	63 million scripts in which 50,000 are picked at random	Alexa top web sites [25]
Perdisci et al. [10]	25,720 distinct malware samples	Different commercial and non-commercial malware sources [32] [33]	25.3 millions of HTTP requests	2 day sniff of a “large and well administered enterprise network”
Cujo [17]	(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 [34]	(1) 200,000 URLs (2) 20,283 URLs	(1) Alexa top web sites [25] (2) Web surfing at authors' institute
Houa et al. [29]	176 DHTML web pages	*StopBadWare* blacklist [35]	965 DHTML web pages	?
JSAND [4]	(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 [36] + local spam trap + Capture-HPC to analyze each URL + manual verification (2) Monitoring “a number of websites” (3) Forums [37] [38] (4) Wepawet	11,215 URLs	Alexa top web sites [25] + most popular queries in Google and Yahoo! search engines + Google Safe Browsing [26] to remove malicious pages
Prophiler [24]	787 web pages	*Wepawet* database + manual inspection	51,171 web pages	Alexa top web sites [25] + Google Safe Browsing API [26] to remove malicious pages
Nunan et al. [8]	15,366 web pages	*XSSed* database [39]	(1) 57,207 web pages (2) 158.847 web pages	(1) *Dmoz* database [27] (2) *ClueWeb09* database [28]
EarlyBird [23]	Same as Cujo	Same as Cujo	100,000 URLs	Alexa top web sites [25]
Revolver [21]	186,032 scripts	Output of an Oracle: Wepawet here	20,732,766 scripts	Output of an Oracle: Wepawet here

Table 3.3: Number and origin of malicious and benign items collected

3.3.2 Choice of features

3.3.2.1 Sets of features of the articles surveyed

All the articles surveyed except [23] and [21] 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 [23] and Revolver [21] are based on existing detectors: respectively Cujo [17] and Wepawet [22] and therefore are dependent on their sets of features.

Table 3.4 sums up the number and type of features chosen for each article.

Article	Features
Likarish et al. [15]	65 features: 50 corresponding to Javascript keywords and symbols + 10 HTML
Cujo [17]	q-gram representation of static code and dynamical sequences of instructions
Houa et al. [29]	171 features: 154 native Javascript functions + 9 HTML + 8 advanced features such as the count of the use of each ActiveX object
JSAND [4]	10 main features in 4 groups: redirection and cloaking, deobfuscation, environment preparation, exploitation
Prophiler [24]	77 features: 19 HTML + 25 Javascript + 12 URL-based + 21 host-based
Nunan et al. [8]	6 features in 3 groups: obfuscation, suspicious patterns and HTML/-Javascript schemes
Perdisci et al. [10]	11 features: 7 statistical + 4 structural

Table 3.4: 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. [29] added features corresponging to ActiveX objects since they are often exposed to vulnerabilities.
Note also that Cujo [17] and EarlyBird [23] 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 [24] and Nunan et al. [8] also refer to URL-based features.

As explained by Houa et al. [29] (see Figure 3.13), 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. [15] 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.

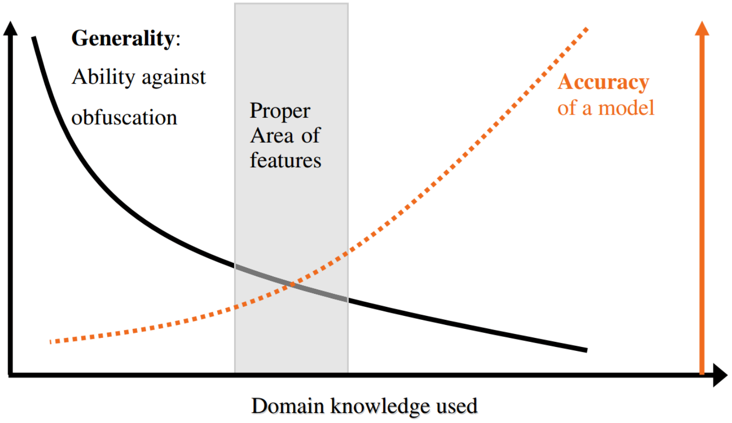


Figure 3.13: Choice of features: Obfuscation vs Accuracy [29]

JSAND [4] employs a set of ten features in 4 groups (see Table 3.5). Environment preparation features (number of bytes allocated through string operations, number of likely shellcode strings) 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. [24] 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 3.5: JSAND features

Furthermore, as highlighted by Canali et al. [24], 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. [10] 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 3.6).

3.3.2.2 Evaluation of features

Some of the articles surveyed state the most useful features for detection. They are recapitulated in Table 3.6.

Likarish et al. [15] 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. [17] 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. [23] 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. [29] 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 3.5). This can be explained by the fact popular attacks always follow the same patterns and sequences.

Article	**Best features**
Likarish et al. [15]	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 [17]	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 [23]	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. [29]	Based on the information gain value: - Distinct word count - Function name - Line count - Word count
JSAND [4]	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 3.6: Set of the most useful features for detection

3.3.3 Classification

While the clustering process depicted by Perdisci et al. [10] and explained in section 4.1 is an instance of unsupervised learning, the other articles surveyed use classifiers to achieve their goals.

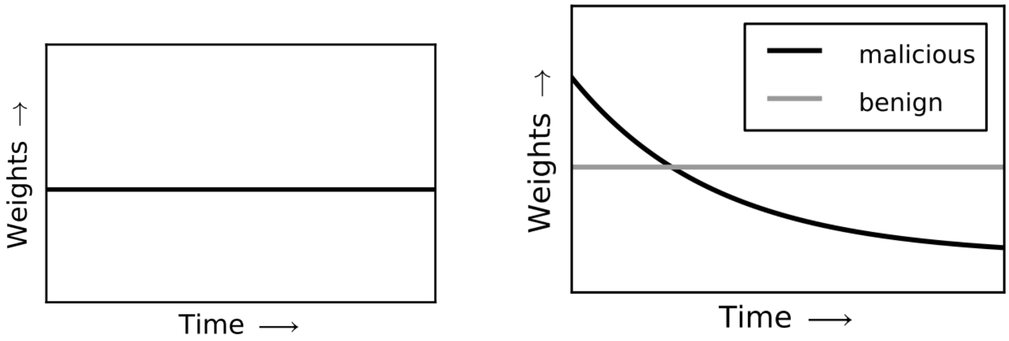
For those approaches, each item is represented by a binary vector of dimension the number of features: if the i^{th} feature is present, the i^{th} dimension of the vector has value 1, and 0 else. From this point, a classifier is used to determine if the item is positive or negative, according to the training dataset. Classifiers used for the surveyed articles are summed up in table 3.7.

Article	**Classifiers**					
	Naive Bayes	**DT & altern.**	**SVM & altern.**	**RIPPER**	**Random Forest**	**Logistic Regr.**
Likarish et al. [15]	x	x	x	x		
Cujo [17]			x			
Houa et al. [29]	x	x	x			
JSAND [4]	x					
Prophiler [24]	x	x			x	x
Nunan et al. [8]	x		x			
EarlyBird [23]			x			

Table 3.7: 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* [40].

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 [23] 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 3.14.



(a) Regular SVM

(b) EarlyBird SVM

Figure 3.14: Temporal weightings for the regular SVM and EarlyBird SVM

Decision Trees 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. [29] 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 [41].
Canali et al. [24] use *Random Forest* which is based on tree bagging and aimed at reducing variance.
Likarish et al. [15] chose an *Alternating Decision Tree* (ADTree) classifier which combines the advantages of decision trees and boosting [42].

RIPPER is a simple rule learning system based on information gain and comprising an incremental reduced-error pruning process [43].

Canali et al. [24] also use classifiers based on *logistic regression* such as J48 or Logistic.

3.3.4 Validation and evaluation

3.3.4.1 Classifier validation

After training a classifier, one has to validate it. A number of approaches [15] [29] [24] [8] 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 3.3) in ten, take 9 out of 10 subsets to train the classifier and leave the 10^{th} subset for validation. Others [17] [23] decided to split their collected dataset and take 75% of it for training and 25% for validation. Statistics are computed from the validation step such as:

- *Precision* or *PPP* (*Positive Predictive Power*) which is equal to: $\dfrac{TP}{TP + FP}$
- *Recall* which is equal to: $\dfrac{TP}{TP + FN}$
- F_2 *score* which is equal to: $\dfrac{5 \cdot Precision \cdot Recall}{4 \cdot Precision + Recall}$
- *NPP* (*Negative Predictive Power*) which is equal to: $\dfrac{TN}{TN + FP}$
- *Accuracy* which is equal to: $\dfrac{TP + TN}{FN + TN + TP + FP}$

where:

- TP is the number of true positive items, that is to say, the number of malicious items labeled correctly.
- TN is the number of true negative items, that is to say, the number of benign items labeled correctly.
- FP is the number of false positive items, that is to say, the number of benign items flagged as malicious ones.
- FN is the number of false negative items, that is to say, the number of malicious items flagged as benign ones.

Four of the articles surveyed use several classifiers and had then to compare their performance.

Likarish et al. [15] 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. [8] 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. [29] 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. [24] 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.

3.3.4.2 Global results of the articles surveyed and discussion

Rieck et al. [17] 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. [8] 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. [29] 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. [15] crawled 24,269 scripts from blacklisted domains [44]. 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. [24] 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. [4] 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 3.3) allowed the authors to manually inspect all of them in order to compute the false negatives. They applied JSAND and other detection tools (ClamAV [45], PhoneyC [46], Capture-HPC [47]) to this malicious dataset and found that JSAND had the best results with a total of only two false negatives.

EarlyBird [23] is an extension of Cujo detector. The same configuration as the work of Rieck et al. [17] 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 [21] 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. [10] 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 3.8. *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 3.8: Signature detection rate on current and future malware samples (1 month training) [10]

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.

3.3.4.3 Validity analysis

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

Cova et al. [4] compared JSAND's false negative rate with ClamAV [45], PhoneyC [46] and Capture-HPC [47], 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. [17] compared Cujo with JSAND [4], 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 [23] was confronted to a regular Cujo occurring over time, one launched at the end of code execution and AVG Anti-Virus [48] 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 [24] with the work of Seifert et al. [49], Ma et al. [50], Feinstein et al. [51] and Likarish et al. [15] 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. [29] 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. [8] compared their work with Likarish et al. [15] 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%.

3.3.5 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 3.9.

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 contributions - Demo Website - the largest amount possible
Features	Semantical	- Obfuscation - Exploitation - Keywords
	Non-semantical	- q-grams
Classification		Compare several classifiers included: - Naive Bayes - SVM - Decision Tree
Validation		Use several metrics included: - Precision - Recall - F_2 score - NPP - 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 3.9: 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 [26]. 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...

TO FINISH

3.4 Implementation choices

Add some figures

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 and classification. Python is indeed a simple and quick language which also implements all the APIs of the tools we use.

3.4.1 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 [52]. A more advanced and dynamic solution is Selenium [53].

3.4.2 `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 authentic user visit on a webpage: both for the time spent on it and the lack of authentic metadata.

3.4.3 Selenium

To make the connection of the detector authentic-ish, there is nothing better than an automated web browser, like Selenium [53]. 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.

3.4.3.1 Selenium WebDriver

Selenium WebDriver [54] allows the user to test a webpages 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 a "real-user time". In addition, the tool allows us to get a bunch of interesting metrics both static and dynamic (features blablablablabla). 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.

3.4.3.2 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. Maybe add an explaining figure?

3.4.4 Classification

We use the Python's module `sklearn` [55] to implement the machine learning part of the project. It is easy to use and allows to choose a large selection of classifiers [56]. In particular, it is very easy to use the SVM (Support Vector Machine) [57]. Example ????

3.4.5 Storing experiments

The solution chosen to store web pages and features values is the NoSQL open-source document database MongoDB [58]. 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.

Chapter 4

Methodology

two parts in each subsection: Architecture/Implementation (Théorie et Pratique)

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 3.3.5. We describe here the methodology used to create such a detector. A global view is depicted by Figure 4.1.

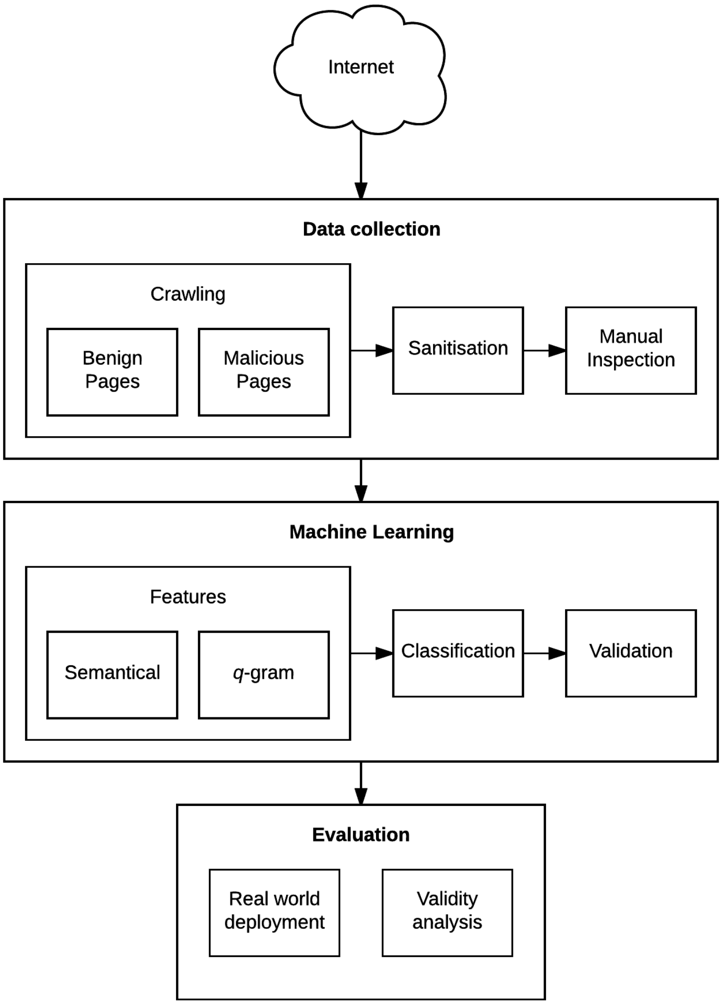


Figure 4.1: Global view of the project

4.1 Data collection

As explained in section 3.3.1, 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.

4.1.1 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 [59].

The dataset must be representative, therefore getting 500 websites is not enough. A good direction is to get URLs linked into the initial webpages with a spider. Briefly, given a list of URLs, a spider looks statically into the respective code and extract new URLs from it. One can then decide the number of levels a spider must explore and grow the list of URLs. In practical, we can use here the spider of ZAP [60], accessible via the open source API [61], to get the list. ZAP, or Zed Attack Proxy, is an OWASP project that has many security features. keep it??

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 [62] 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 remove crawled URLs contained in Google Safe Browsing's blacklist [26]. 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.

4.1.2 Malicious dataset

It is more difficult to collect malicious items that are still used in the wild. A number of blacklists exist like MalwareDomains [44] and MalwareDomainList [37]. 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 determine whether a webpage is still accessible "normally" and does not lead to an error code (e.g. 403, 404, 502, 503).

Eventually, 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 use either for training our classifiers or for evaluating the detector's performance.

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

4.2 Extraction of features

We compare two different types of features (see section 3.3.5) for our experiment. The so-called semantical features – features that have a real meaning, and non-semantical features. Among those two types of features, we distinguish static features from dynamic features and we use different approaches to extract them.

4.2.1 Semantical features

Static features are isolated using a parser. A good example is YACC grammar [18], or its GNU version Bison [63], which are a LALR parser generators that split the webpages according to certain rules. However, simple metrics like the number of words, lines, characters etc. do not require such grammars and can be easily computed.

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 [64] 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.

4.2.2 *q*-grams

Don't have the time to try: future work

q-grams are extracted from the webpages after a few steps summed up in Figure 4.2. 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.

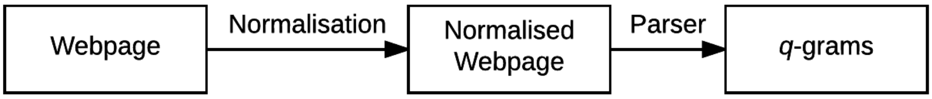


Figure 4.2: Turning a webpage into a q-grams

4.2.3 Machine Learning

It is very interesting to be able to compare the results of diverse classifiers on our datasets. The Python's module `sklearn` allows us to try a large panel of them.

The first draft of the detector use the SVM (Support Vector Machine) classifier to classify our data.

Chapter 5

Implementation

After depicting the methodology used to build the detector, represented by Figure 4.1, let us focus on its implementation. Figure 5.1 shows the basic technical architecture of the detector whereas Figure 5.4 refers to the advanced architecture using Selenium Grid (refer section XXXXX).

5.1 Basic

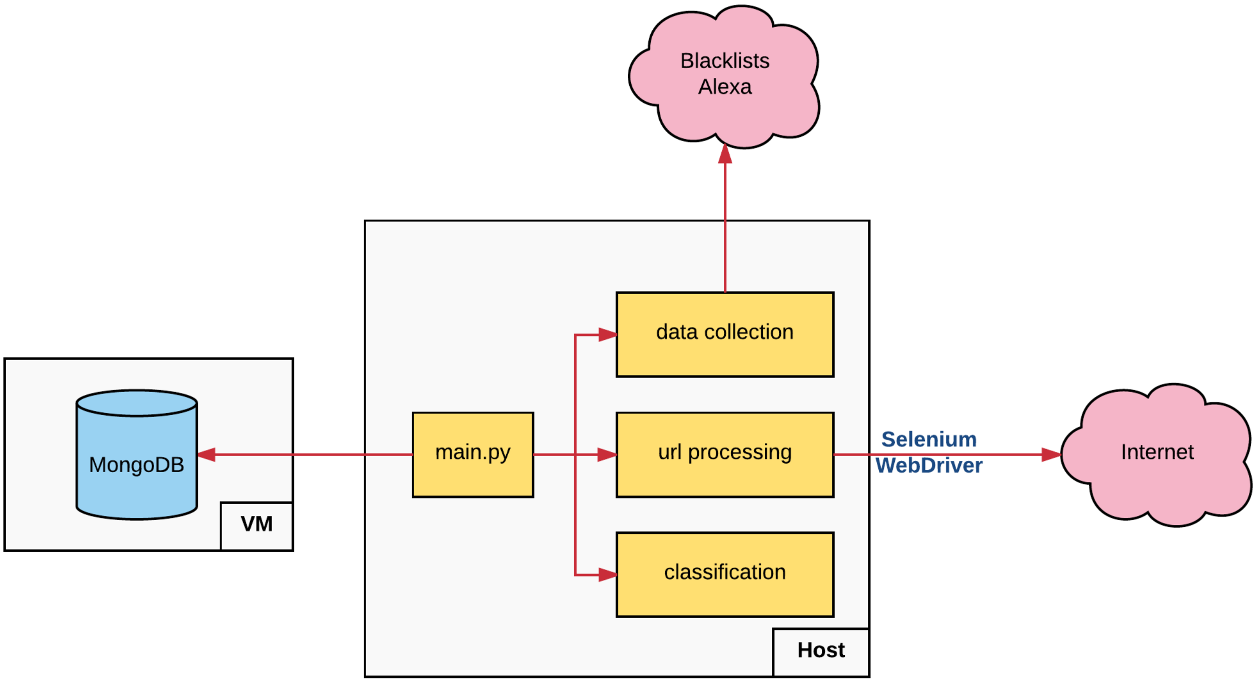


Figure 5.1: Basic Architecture

The main script (`main.py`) calls successively the data collection, URL processing and classification functions. Those pieces of code constitute the skeleton of the detector. They are extensively explained hereinbelow.

5.1.1 Main

5.1.2 Data collection

Add diagram for alexa-crawl and mw-crawl - crawl-mwd - crawl-mwdl

As mentioned in section XXXXXX, we collect the most popular websites as our benign dataset. Most popular websites can be found for free on Alexa [25]. The website provides a charged API to get them easily. For this project, we wanted to get the pages for free. Therefore, given that we can get the 500 most popular URLs for free but not access them with an API, we created a simple script to fetch the URLs from the website and store them in a text file. This is what alexa-crawl() ??? Appendix or diagram ? does.

Malicious websites are found on MalwareDomains [44] and MalwareDomainList [] 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 flux giving the latest malicious URLs found. A simple parser allows us to get those URLs. Both sources let us access to 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.

Future: automated script

5.1.3 URL processing

- Pseudo-code / python code? for benign and malicious

```
# Setup
vm_url = ’146.169.47.251’
db_port = 27017
client = MongoClient(vm_url, db_port)
db = client.projectDB
db_urls = db.urls
benign_urls_addr = ’PATH/MSc-project/Crawler/alexa-top500’
malicious_urls_addr = ’PATH/MSc-project/Crawler/malwaredomains-raw-recent’

# Data collection
crawl_benign()
crawl_malicious()

# URL processing
main_benign()
main_malicious()

sanitize_db(db_urls)

# Classification
ml()

# Stats
```

Figure 5.2: Python pseudo-code of the main program

```
- 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 5.3: Pseudo-code of URL processing functions `main_benign()` and `main_malicious()`

5.1.4 Classification

- Issue: numerical features with SVM
- Cross validation: cross val predict [65] vs cross val score [66]
- Confusion matrix TP FP TN FN
- Feature engineering for name of most frequent word

5.2 Grid

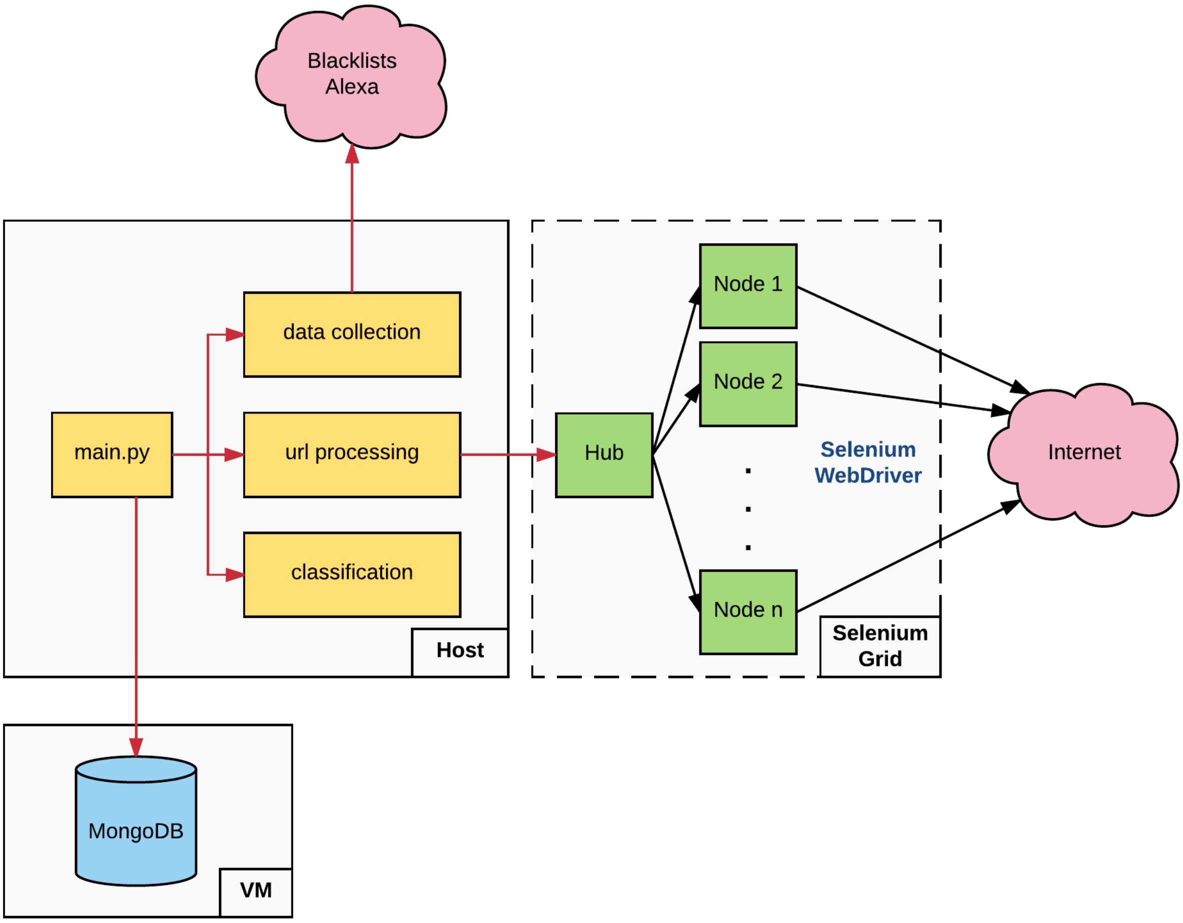


Figure 5.4: Grid Architecture

Chapter 6

Experiments and results

6.1 Data collection and performance

- Distribution of url analysed time
- Number of sanitized URLs

Number of benign URLs	453
Number of malicious URLs	277
Total analysed time	36 minutes
Mean analysed time	3 seconds
Classification time	9 minutes

Table 6.1: Performance measurements

6.2 Features

- Line count
- Letter count
- Word count
- `eval` count
- Percentage of whitespace
- Most frequent word

6.3 Validation measurements

Cross Validation

Add specificity in section XXXXXXX Background

Accuracy	82%
Precision	0.93%
Recall	90%
F_2 score	88%
NPP	80%
Specificity	70%

Table 6.2: Validation measurements

6.4 Comparison of classifiers

- Decision Trees

6.5 Comparison of features

- LDA
- try one by one

6.6 Real-world deployment

6.7 Comparison with other approaches

Chapter 7

Conclusions and future work

Future work:

- Automated script to get the malicious script from the different sources.
- Compare different browsers: HtmlUnit, Chrome, Firefox etc.
- q-grams
- More classfiers to compare

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

[4] 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.

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

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

[7] 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.

[8] 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.

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

[10] 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.

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

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

[13] W. Xu and F. Zhang and S. Zhu. The Power Of Obfuscation Techniques In Malicious JavaScript Code: A Measurement Study. pages 9–16, Oct 2012.

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

[15] 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.

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

[17] 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.

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

[19] 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.

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

[21] 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.

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

[23] 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.

[24] 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.

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

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

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

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

[29] 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.

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

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

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

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

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

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

[36] Spamcop. https://www.spamcop.net/.

[37] MalwareDomainList Forum. http://malwaredomainlist.com.

[38] Milw0rm Forum. http://milw0rm.com.

[39] XXSed database. http://www.xssed.com.

[40] 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.

[41] Lecture on Boosted Decision Trees by Christian Autermann, university of hamburg. http://wwwiexp.desy.de/users/auterman/talks/20070706_hh_bdt.pdf.

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

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

[44] MalwareDomains. http://www.malwaredomains.com.

[45] ClamAV Antivirus. https://www.clamav.net/.

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

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

[48] AVG Antivirus. http://www.avg.com/.

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

[50] 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.

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

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

[53] Selenium Documentation. http://www.seleniumhq.org/.

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

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

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

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

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

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

[60] OWASP ZAP. https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project.

[61] OWASP ZAP API Repository. https://github.com/zaproxy/zaproxy/wiki/ApiDetails.

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

[63] GNU Bison. https://www.gnu.org/software/bison/.

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

[65] sklearn's Documentation of the function cross_val_predict. http://scikit-learn.org/stable/modules/generated/sklearn.cross_validation.cross_val_predict.html.

[66] sklearn's Documentation of the function cross_val_score. http://scikit-learn.org/stable/modules/generated/sklearn.cross_validation.cross_val_score.html.