An Empirical Study of Metric-Based Methods to Detect Obfuscated Code

Corrado Aaron Visaggio, Giuseppe Antonio Pagin and Gerardo Canfora

Department of Engineering, University of Sannio
82100 Benevento, Italy
visaggio/pagin/canfora@unisannio.it

Abstract

Protecting data and applications from malware and other forms of malicious code has assumed a great relevance in the current era of pervasive web-based applications. Attackers often use code obfuscation to hide harmful programs from automatic detection. Several researchers have proposed methods to classify an unknown program as malicious or benign; however, little work has been done to identify obfuscated code. A promising approach to detect obfuscated code consists of using a set of metrics, collected by static analysis, to classify a program.

In this paper we present an empirical evaluation of three text-based metrics to identify obfuscated code. Our experiment shows that the effectiveness of these metrics depends on the obfuscators: there are cases in which the metrics allow the proliferation of false positives (i.e., misclassification of clear code as obfuscated code), which is bothering but not dangerous, and cases where false negatives (i.e. misclassification of obfuscated as clear code) proliferate, which is definitely more dangerous. Based on our experiment, we propose a combination of these three metrics and show how this combination outperforms the individual metrics.

Keywords: *malicious code, obfuscated code, static analysis*

1. Introduction

Code obfuscation consists of transforming the code of a program into a semantically equivalent form that is harder to understand for a human being, or to be analyzed by static analysis tools [1]. Code obfuscation can be used for dual purposes: attack and defense. As a matter of fact, code obfuscation can be aimed at protecting source code from tampering, information leakage (source code might contain secrets, such as credentials or details of a confidential protocol) or illegitimate copies. From the attacker's perspective, code obfuscation is largely applied to camouflage malware, like worms and viruses, impeding antivirus to recognize their signatures.

JavaScript is a scripting language that allows developers to add sophisticated features to their web applications. The dynamic nature of JavaScript and its tight integration with the browser make it hard to detect and block malicious JavaScript code; thus, it is often used to perform attacks.

Attacks that target web clients are becoming pervasive [23]. Many common attack methods use malicious JavaScript, including cross-site scripting [24] and web-based malware distribution. JavaScript is used to redirect a user to a website hosting malicious software, to create a window recommending to user download a fake codec, to detect what software versions the user has installed and select a compatible exploit or to directly execute an exploit.

Drive-by downloads is a common and insidious form of attacks [10]. In a drive-by download attack, a victim is lured to a malicious web page. The page contains code, typically written in JavaScript, that exploits vulnerabilities in the user browser or in the browser's plugins. If successful, the exploit downloads malware on the victim machine, which, as a consequence, often becomes a member of a botnet.

The use of obfuscators in the current web based scenario is widely expanding, especially for malicious purposes. In order to avoid obfuscated malware to hurt, an antimalware system must preliminarily recognize the presence of obfuscated code, and then it can assess whether the code is harmful or not. The literature offers several mechanisms to de-obfuscate code [4, 9, 14], and for recognizing whether the code is harmful or not [2, 11, 16, 17], but there is not much research that addresses the automatic detection of obfuscated code.

Obfuscation can be used also for legitimate purposes, such as protecting intellectual property, and thus detecting obfuscated code does not necessarily entail that a malware has been discovered. However, as many attackers use obfuscation to hide their malicious code, detecting obfuscated code is a preliminary step to then recognize whether, or not, the obfuscated code is malicious.

The goal of our research is to study effective mechanisms, based on static analysis, for automatically detecting that a JavaScript code has been obfuscated. Static analysis is usually fast and can be performed during the download of a page, thus preventing users to be victim of malware. Once an obfuscated piece of code has been detected, it can be de-obfuscated and classified as malicious or benign with one of the existing techniques [2, 11, 16, 17]. Our work focuses on identifying obfuscated code: we do not address the problem of recognizing maliciousness, neither we propose solutions for de-obfuscating the code.

In this paper, we analyze, through experimentation, the advantages and drawbacks of three text-based metrics, introduced in [5], and propose and validate a new metric for the same purposes. The detection method consists of computing metrics that reflect characteristics of the code text, as well as the distribution of the characters or the occurrences of special characters, and compare metrics against thresholds. The assumption is that each metric evaluates an aspect of the text that could be an indicator of obfuscation. When a metric overcomes a certain threshold, the code is likely obfuscated.

The metrics considered are: n-gram, which counts the occurrence of a particular set of characters within code strings; entropy, which evaluates the bytes distribution corresponding to a string; and wordsize, which measures the length of the code strings.

We pose three research questions:

- Comparison of the three metrics: are the three metrics equivalent in terms of effectiveness at revealing obfuscated JavaScript code?
- Independence from the obfuscation technique: does the effectiveness depend on the obfuscation technique used?
- Combination of the three metrics through a linear function: can a combination of the three metrics be more effective than the individual metrics in detecting obfuscated code?

The main contribution of this work is threefold:

- To explore a topic, the automatic detection of obfuscated code, which is an essential part of the protection against obfuscated malware. There is a need for mechanisms which could detect obfuscated code on the fly, for instance during the loading of a web page which contains harmful obfuscated code, in order to block it before execution.

- To strengthen the existing empirical evidence of the effectiveness of metric-based detection of obfuscated code. Replications of experiment and empirical evidence are the basis for building a body of knowledge with the scientific method.

- To propose and validate a new metric which could be more effective than existing metrics. The new metric is a linear combination of the three metrics: n-gram, entropy and wordsize.

The paper is organized as follows: Section 2 discusses related work, and section 3 illustrates the metrics. Section 4 introduces the experiment and discusses the results. Finally, conclusions are drawn in Section 5.

2. Related Works

The literature presents many works concerning de-obfuscation [4, 9, 14] and malware detection [2, 11, 16, 17, 30]. Behind hiding malware, code obfuscation is used for legitimate purposes, such as protecting intellectual property or enforcing security [18, 12].

In this section we focus on the papers that address the issues related to the automated detection of obfuscated code.

Obfuscated code is often analyzed manually, i.e. by executing the suspicious code. In the case of JavaScript, a common approach consists of using a JavaScript engine to execute the suspicious code by replacing dangerous actions with simple print instructions, thus revealing possible malicious code [22]. The JavaScript engine must, of course, be run out of a browser to prevent any exploitation. Tools such as Rhino [21], SpiderMonkey [13] and V8 [15] are widely used to de-obfuscate code. There are some JavaScripts that are effective on some browsers and ineffective on others. Rhino does not need any web browser for executing JavaScript code, while SpiderMonkey and V8 have been designed to work with Mozilla Firefox and Google Chrome, respectively. These engines are often limited by the fact that they do not completely implement the browser context, such as the DOM or other APIs. It is not rare that a de-obfuscation task halts on an unknown variable that is in fact a DOM variable or an API call. Analysts often have to declare statically the missing context in order to pursue analysis. Obviously, manual analysis is time-consuming and tedious, and researchers often resort to self-developed tools to automate such task. Malzilla [24] is an open source project that offers some functionality to explore malicious webpages and view their code.

Until recently, the topic of obfuscation in web scripting languages, and in particular JavaScript, was seldom discussed in the academic community. Researchers used to present works in which they assumed that malicious code would be available as plain text, which is now rarely the case. In reaction to this observation, some researchers have studied the obfuscation problem and developed machine-learning-based systems to detect obfuscated code, assuming that obfuscation could be, in some cases, an indicator of malice [19], as it is used to impede the antimalware scanners to recognize the code signature. A more straightforward study has proposed to learn all patterns of obfuscated malicious scripts, assuming these are in a finite number [6].

At the best knowledge of the authors, very little work has been done in the direction of automatic detection of obfuscated code. We recall that the solutions able to intercept harmful code are necessary but not sufficient for a robust protection. It is important to first recognize automatically when a code is obfuscated, as an obfuscated program could bypass the checks of signature-based antivirus.

3. Metrics for Detecting Obfuscated Code

This section introduces the three metrics that are object of our experimentation; these metrics have been introduced in [5].

3.1. N-gram

This metrics counts the frequency of ascii code occurrences within the string. The ascii codes are classified in three categories: alphabet, number and special characters. We will focus only on special characters as most of the obfuscated strings make excessive use of specific characters like | , [,], @, x, u, and so forth. The assumption is that an obfuscated code string makes use of a subset of special characters in a more intensive way than other code strings.

3.2. Entropy

Entropy measures the bytes distribution within the string, and it is computed as follows:

$$E(B)=-\sum_{i=1}^{N}\left(\frac{bi}{T}\right)\log\left(\frac{bi}{T}\right), \quad (1)$$

where B= b_i, i= 0,1,…,N and T= $\sum_{i=1}^{N} bi$

In equation (1), b_i is the value corresponding to the occurrence of each byte, while T is the total number of bytes within the string. The less the bytes within the string, the more E tends to 0. The max value of E is LogN, meaning that the bytes are distributed around this set of bytes. N is set to 256 because there is the need to read all the possible ascii characters.

This metric divides the code strings into two categories: the first category includes common (and not suspected) code, the second category includes strings with a high number of special characters (and thus the most suspected of being obfuscated code). Reference [5] suggests the value 1.2 as a threshold for establishing if a string is obfuscated (entropy<1.2) or not (entropy>1.2). There is then a third category, which includes safe cases, and is characterized by values of entropy much greater than 2.

3.3. Wordsize

As the instructions of a JavaScript code are supposed to be read by humans, their length is usually under a certain threshold (the brain has a limited capability of processing symbols, as well known by cognitive literature). Even if an instruction length has a wide variability, one may expect that a JavaScript statement does not require more than 200 bytes. This metric is the number of bytes in a word, where a word corresponds to an instruction. The assumption is that an obfuscated string tends to have a greater length.

4. The Experiment

The experiment aims at understanding the effectiveness of the three proposed metrics in detecting obfuscated code. More precisely we analyzed three facets of the question, declined in three research questions:

Q1: are the three metrics equivalent in terms of effectiveness?

The first target of our investigation is the effectiveness of the metrics in recognizing the obfuscated code. Additionally, we wonder whether these three metrics are equivalent, or there are cases where one is better than the others. N-gram based detection could fail in some cases as there are non-obfuscated instructions which require special characters. On the other hand, there are some programming languages, like Perl, which require large use of special characters. For a similar reason, there could be cases in which the occurrences of special characters within a certain portion of code could bring about a low value of entropy but being normal. Both the cases would lead to a proliferation of false positives.

Q2: does the effectiveness depend on the obfuscation technique used?

The second target of our investigation concerns the independence of detectors performances from the specific obfuscator used. Do the metrics vary their effectiveness for different obfuscators, or are they robust with respect to the differences of techniques used for obfuscating code?

Q3: is a combination of the three metrics more effective than the single metric in detecting obfuscated code?

We investigate if a combination of metrics could compensate the limitations of the individual metrics and provide for a more effective tool for detecting obfuscated code. Foe example, if an obfuscator produces a few special characters, as it uses an alphanumeric coding, an n-gram based detector could fail, but a wordsize-based detector could succeed as usually these kinds of obfuscators produce very long strings, as they repeat frequently the same set of characters.

4.1. Experiment Organization

To investigate research questions Q1, Q2 and Q3, we have extracted 1000 JavaScript scripts from 1000 web pages, randomly selected in the Internet. The selection was organized so that the sample contains both malicious code and innocuous code, in order to have a highly randomized sample for the experiment.

These scripts have been obfuscated by using four different obfuscators, available also as web applications [26, 27, 29, 28], respectively indicated with Obf1, Obf2, Obf3 and Obf4, for brevity sake, and metrics have then been computed on the strings produced by the obfuscators.

Obf2 applies control obfuscation by inlining code [3, 8], whereas the other obfuscators apply layout obfuscation. The former changes the way the statements are grouped, by replacing a calling function with the body of the function itself. Layout obfuscations act on code information that is unnecessary to its execution (an example is the Java obfuscator Crema [25]), for instance by formatting source code, variable names and comments. These obfuscations are typically trivial and aim at reducing the amount of information available to a human reader. Layout transformations include the removal of comments and the change of identifiers. For example, by replacing identifiers of methods and variables with meaningless identifiers, any information on the functionality of a method or on the role of a variable is removed. Scrambling identifier names is a one-way transformation that does not affect execution.

In summary, five sets of 1000 strings have been generated: one set of not-obfuscated strings corresponding to the JavaScript scripts extracted from the Web and four sets of obfuscated strings. Each string had the correspondent obfuscated versions obtained by running the four obfuscators.

To help the reader understand the differences among the four obfuscators, we propose the four versions for the same non-obfuscated script:

```
document.write('<span id="topmsg" style="position:absolute;
visibility:hidden">'+message+'</span>')
```

Obf1 produces:

```
var_0x24c8=["\x3C\x73\x70\x61\x6E\x20\x69\x64\x3D\x22\x74\x6F\x70\x6D\x73\x6
7\x22\x20\x73\x74\x79\x6C\x65\x3D\x22\x70\x6F\x73\x69\x74\x69\x6F\x6E\x3A\x6
1\x62\x73\x6F\x6C\x75\x74\x65\x3B\x76\x69\x73\x69\x62\x69\x6C\x69\x74\x79\x3
A\x68\x69\x64\x64\x65\x6E\x22\x3E","\x3C\x2F\x73\x70\x61\x6E\x3E","\x77\x72\x
69\x74\x65"];document[_0x24c8[2]](_0x24c8[0]+message+_0x24c8[1])
```

Obf2 produces:

```
eval(function(p,a,c,k,e,d){e=function(c){return(c<a?":e(c/a))+String.fromCharCode(c
%a+161)};if(!".replace(/^/,String)) {while(c--) {d[e(c)]=k[c]|| e(c)}k= [function(e)
{return d[e]}];e=function(){return'\ [\xa1-\xff]+'} ;c=1}; while(c--
){if(k[c]){p=p.replace(new RegExp(e(c),'g'),k[c])}}return
p}('.(\'<=""=:;:">\'++\'</>\')',11,11,'span|topmsg|id|write|document|style|position|messa
ge|hidden|visibility|absolute'.split('|'),0,{}))
```

Obf3 produces:

```
document.write%28%27%3Cspan%20id%3D%22topmsg%22%20style%3D%22positi
on%3Aabsolute%3Bvisibility%3Ahidden%22%3E%27+message+%27%3C/span%3E
%27%29
```

Obf4 produces:

```
646F63756D656E742E777269746528273C7370616E2069643D22746F706D7367222
07374796C653D22706F736974696F6E3A6162736F6C7574653B7669736962696C6
974793A68696464656E223E272B6D6573736167652B273C2F7370616E3E2729
```

The differences among the four obfuscators are evident. Obf1 makes large use of the special character "\" and repetitions of similar strings of characters like "\x". This should produce high values for n-gram metric and, as it produces unusually long strings, this obfuscator should be detected by wordsize based techniques. Entropy should be high too, due to the high occurrences of uncommon set of characters.

Obf2 could hide the obfuscated code to a not expert human eye; as a matter of fact, the obfuscated string could appear, at a first glance, a "normal" code string. If we observe it closer, we can notice that the string is very long and contains unusual sequences of characters, like "\[\xa1-\xff]+'};". Finally, the instruction is written in a form too complex for being part of a readable code. Therefore, the metric that could success detecting this obfuscating is entropy.

Obf3 produces strings that are not too long, so they could bypass a wordsize-based control, but it brings about too many sequences of unusual characters, thus should be detected by an entropy-based control and an n-gram-based control. Obf4 does not use special characters, but only hexadecimal codes, thus the metric n-gram and entropy should be ineffective. It produces very long strings, and this should help the wordsize-based obfuscators to succeed.

The three metrics n-gram, entropy, and wordsize were used to characterize the code samples in order to establish if the analyzed code is obfuscated or not. This will help to answer the research questions Q1 and Q2. In order to answer Q3, we generated three new metrics by aggregating through a weighted sum the three metrics. The aggregated metrics and the related analysis of data are discussed in a next subsection.

4.2. Descriptive Statistics of Results

The raw data were collected with a software tool developed by the authors of the paper and which computes the values corresponding to the three metrics for each string.

4.2.1. N-gram: The box plots of the observations gathered with the n-gram-based detector are showed in Figure 1. It emerges that the number of special characters found in the non-obfuscated code is not so smaller than the one used for the sets of the obfuscated versions of the strings. The first two obfuscators have a number of special characters largely greater than the original code. But the same thing cannot be said for the last two obfuscators. This suggests that a population of code usually can have a limited set of "unusual" characters (which is a tautology: as a matter of fact a character is "unusual" just because it rarely occurs in the source code). The number of unusual characters seems to vary a lot from case to case. The fifth set represents a special case, as there is not any special character in the string. This is due to the coding used by the corresponding obfuscator. Definitely n-gram is not able to discriminate obfuscated code from not-obfuscated code in all the possible cases. In fact, the results obtained with the first two obfuscators indicate that there is a considerable increment of special characters within the strings. Conversely, this does not happen with the third one, which produces a number of special characters not much different from the one we find into the original code. A serious anomaly is represented by the fourth obfuscator, which has 0 unusual characters.

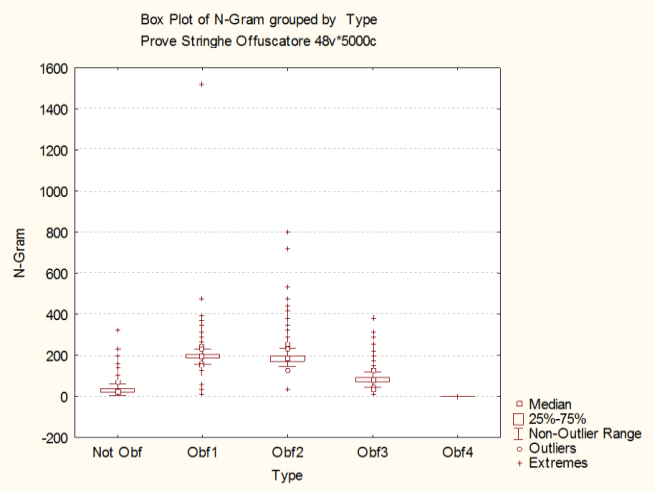

Figure 1. Box Plots of n-gram-computed for Non-obfuscated Code and the Code

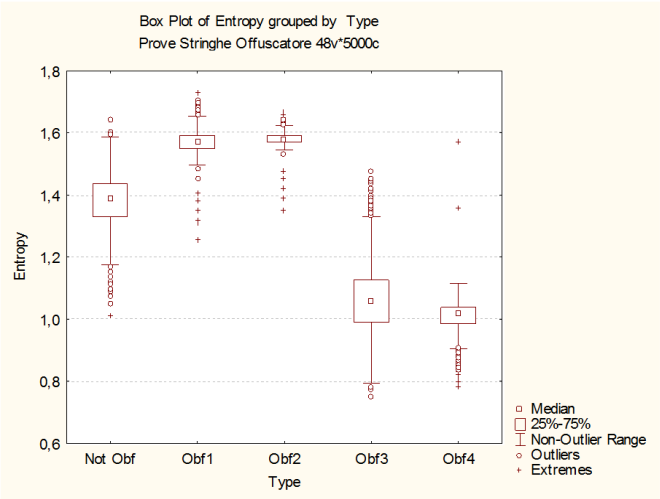

Figure 2. Box Plots of Entropy-computed for Non-obfuscated Code and the Code

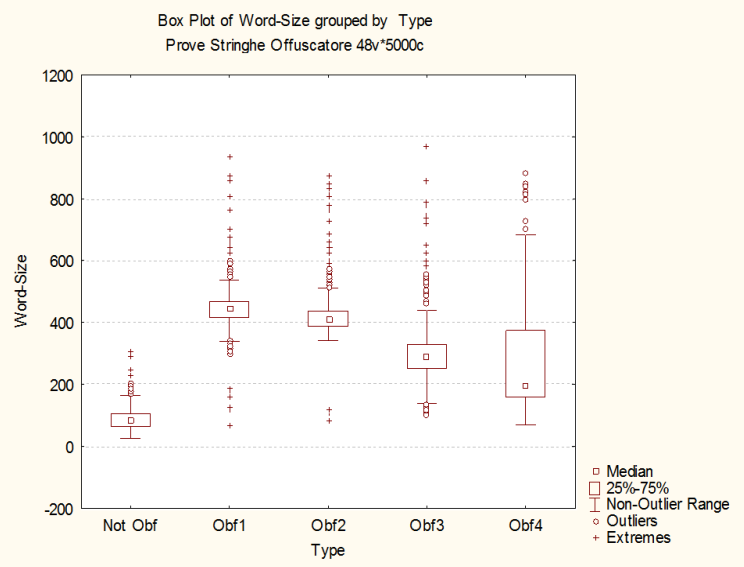

Figure 1. Box Plots of Wordsize-computed for Non-obfuscated Code and the Code Obfuscated with the Four Obfuscators

4.2.2. Entropy: The results produced by the entropy-based detector are showed in Figure 2. As expected, the median value of entropy for the non-obfuscated string is about 1.4, pretty close to 1.2, which is the value assumed as a threshold for detecting the obfuscated code. We recall that the entropy increases with the increasing of diverse characters occurring within the string. As a matter of fact, the fourth and the third obfuscator produced very low values for this metric because, due to the particular obfuscation techniques used, the same set of characters is repeated for all the strings.

A first observation is that the obfuscators greatly affect the effectiveness of entropy-based detection. The first and the second obfuscator, in fact appear much weaker than the others, as

they produce values that are much higher than the original code and the other obfuscators. The internal variability of the samples (*i.e.*, the width of box-plots) is small. Variability plays an important role in detection activities, as it is related to predictability. If the values of a phenomenon vary in a smaller range the detection is more reliable. Finally, the entropy of the third and fourth sample is much lower than the not-obfuscated code, which reveals the ineffectiveness of entropy-based detector with this kind of obfuscation mechanism. Entropy could be not considered a strong system of detection for obfuscated strings.

4.2.3. Wordsize: The box plots of wordsize-based detector are reported in the Figure 3. The median of the sample corresponding to the original code is much lower than the medians of the other sets, and also the sample variability is smaller, which suggests that the wordsize metric could be reliable for detecting obfuscated code. Wordsize is effective also for the fourth obfuscator, which was not detected by the other two metrics. The only limitation of wordsize is that the fourth case shows a strong variability of the sample, which could deteriorate the detection precision.

4.3. Analysis of Aggregated Metrics

The previous section has highlighted the strength and limitations of the three metrics n-gram, wordsize and entropy, in detecting obfuscated code. Their effectiveness largely depends on the code that is going to be obfuscated or the obfuscation method used. Thus we investigated whether a metric that linearly combines the three metrics could mitigate the limitations of each metric while emphasizing their ability to detect obfuscated code. We have explored three combinations, in the form of weighted sum:

$$fi= a1*N\text{-}gram + a2*Entropy + a3 * WordSize.$$

In particular, we used the following weights:

$$f1=0.4*Entropy + 0.4*N\text{-}gram/10 + 0.2 * WordSize/100 \quad (2)$$
$$f2=0.4*Entropy + 0.2*N\text{-}gram/10 + 0.4 * WordSize/100 \quad (3)$$
$$f3=0.2*Entropy + 0.4*N\text{-}gram/10 + 0.4 * WordSize/100 \quad (4)$$

As the three metrics produce numerical values that differ about one order of magnitude, for n-gram and wordsize we used a normalization factor of, respectively 10 and 100. Box plots of results for the three linear combinations are illustrated in Figures 4, 5, 6. The three combined metrics present approximately the same characteristics: the variability of each sample is much reduced, the obfuscators have similar performances, and there are not samples with anomalies as in the case of n-gram for the fourth obfuscator.

Let's analyze each case. All the three aggregate metrics show a variability of samples variance that does not differ much among the different samples. The variability is very tight in all the samples and much smaller than the variability of the not-obfuscated code. And this is a relevant property for a detector, as it fosters the precision of detection.

All the samples corresponding to the four obfuscators have box plots that are close among each other and far from the box plots of the non-obfuscated code. This does not happen for all the initial metrics, especially for entropy, where the third and the first obfuscator produce values that are smaller than the non-obfuscated code, with the side effect of producing (dangerous) false negatives. The further advantage of the new aggregate metric is that it does not produce useless samples, as the obf4 sample produced by the n-gram-based detector, where all the values are 0s.

Each one of the aggregate metrics seems to be preferable to individual metrics to identify obfuscated code, with a preference for f3, which emphasizes the differences among the obfuscated codes and the non-obfuscated code, de-emphasizes the differences among the different obfuscators (*i.e.*, the effectiveness of detectors does not vary much by varying the obfuscation method) and produces samples with the lower variability (*i.e.*, detection is more accurate). As the values of the samples are lower than the values of the original code, these new functions express a threshold under which the code is obfuscated.

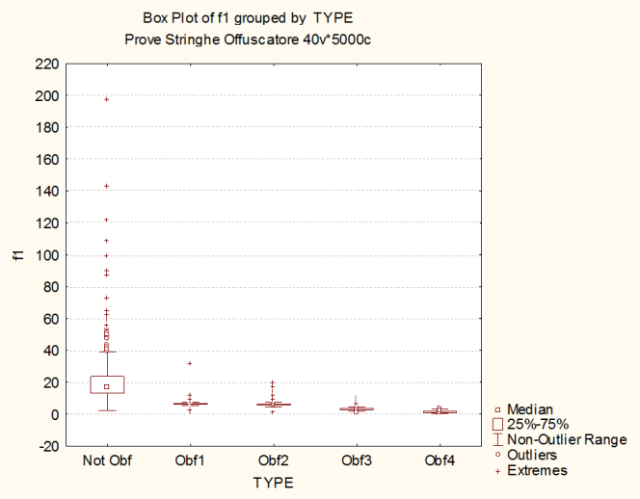

Figure 2. Box Plots of f_1 Computed for the Non-obfuscated Code and the Code Obfuscated with the Four Obfuscators

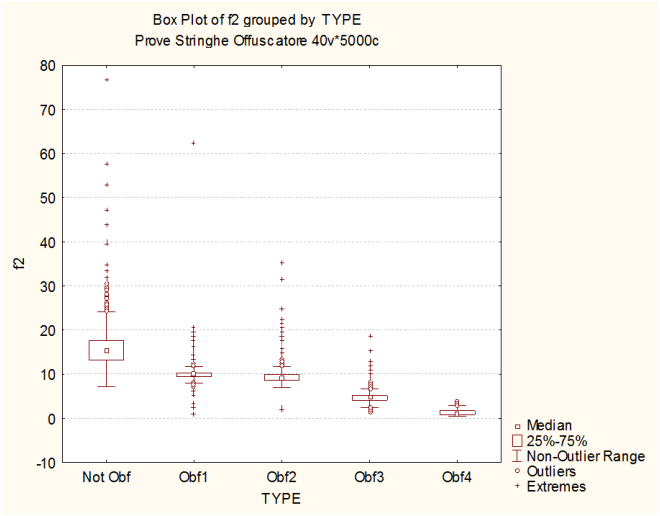

Figure 3. Box Plots of f_2 Computed for the Non-obfuscated Code and the Code Obfuscated with the Four Obfuscators

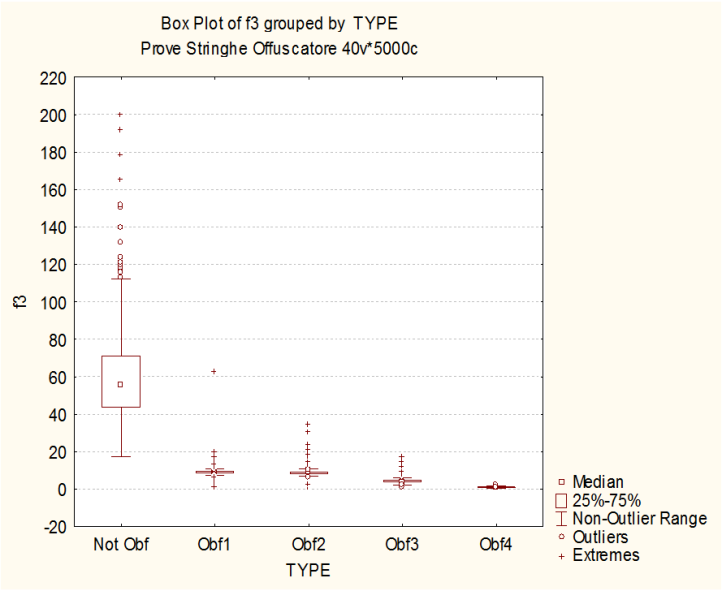

Figure 4. Box Plots of f_3 Computed for the Non-obfuscated Code and the Code Obfuscated with the Four Obfuscators

4.4. Test of the Hypotheses

We performed Mann-Whitney tests (p-value fixed at .05) and Kolmogorov-Smirnov (p-value fixed at .01) for the following hypotheses, in order to obtain empirical evidence with regards to the results gathered to answer Q1 and Q2. Mann Whithney test is used to verify whether the values of a continuous distribution belong to the same population. The null hypothesis of the Mann-Whithney test is that the two samples belong to the same population and, thus, their distributions of probability are equal. The test of Kolmogorov-Smirnov is a valid alternative to the Mann Whithney test, especially for small samples. We chose to run both the tests in order to enforce the external validity of our findings.

H_{01} Are the values of metrics n-gram, entropy, and wordsize for the obfuscated code different from the non-obfuscated code?

H_{02} Are the values of metrics n-gram, entropy, and wordsize different for the different obfuscators?

The tests were run by comparing the values obtained for the non-obfuscated code and the obfuscated code produced with the four obfuscators. Table 1 shows the results of Mann-Whithney ("Ma.Whi." in the correspondent columns) and Kolmogorov-Smirnov ("Kol.Smi." in the correspondent columns). In the first row of the table we compare the value of the metric corresponding to the non-obfuscated code ("NotObf" in the table) with each obfuscated code produced by the four obfuscators (Obf1, Obf2, Obf3, and Obf4 in the table). This test concerns the H_{01} hypothesis. The following rows show the comparison between each obfuscated code with the obfuscated code produced by the other obfuscators. These tests concern the H_{02} hypothesis.

The values were computed with the three detectors, based on n-gram (Table 1-A), Entropy (Table 1-B), and wordsize (Table 1-C). When the p-level is much smaller than 0.00001 for brevity sake we write "p<0.00001", otherwise we write the exact value.

In order to gather evidence on the research questions Q3, H_{01} and H_{02} were tested on the samples produced by the detectors based on the aggregated metrics f_i. Also in this case, the Mann Whithney tests were run and p-value fixed at .05, while the Kolmogorov-Smirnov tests had the p-value fixed at .01 (results are showed in Table 2). The hypotheses can be rejected for all the metrics n-gram, wordsize and entropy, but the limitations regarding these three metrics should be kept in mind. We gathered empirical evidence that all the three metrics are able to produce differences among obfuscated code and non-obfuscated code, and that the values produced by single obfuscators are different. The two null hypotheses could be rejected also for the new combined functions.

We collected evidence that the new functions are able to discriminate obfuscated code, without the limitations of the single metrics. Additionally, different obfuscators produce different values of the metrics, thus the values that are produced depend on the obfuscator used. This limitation is mitigated by the fact that the new functions can produce samples with better values.

Table 1. P-value of Hypotheses Testing (Mann-Whithney & Kolmogorov-Smirnov) for N-gram (A), entropy (B) and wordsize (C)

Hyp.	N-GRAM	Obf1		Obf2		Obf3		Obf4	
		Ma. Whi.	Kol. Smi.	Ma. Whi.	Kol. Smi.	Ma. Whi.	Kol. Smi.	Ma. Whi.	Kol. Smi.
H_{01}	NotObf	P<.00001	P<.00001	P<.00001	P<.00001	P<.00001	P<.00001	P<.00001	P<.00001
H_{02}	Obf1			P<.00001	P<.00001	P<.00001	P<.00001	P<.00001	P<.00001
H_{02}	Obf2					P<.00001	P<.00001	P<.00001	P<.00001
H_{02}	Obf3							P<.00001	P<.00001

(A)

Hyp.	ENTR.	Obf1		Obf2		Obf3		Obf4	
		Ma. Whi.	Kol. Smi.	Ma. Whi.	Kol. Smi.	Ma. Whi.	Kol. Smi.	Ma. Whi.	Kol. Smi.
H_{01}	NotObf	P<.00001	P<.00001	P<.00001	P<.00001	P<.00001	P<.00001	P<.00001	P<.00001
H_{02}	Obf1			P<.00001	P<.00001	P<.00001	P<.00001	P<.00001	P<.00001
H_{02}	Obf2					P<.00001	P<.00001	P<.00001	P<.00001
H_{02}	Obf3							P<.00001	P<.00001

(B)

Hyp.	WS	Obf1		Obf2		Obf3		Obf4	
		Ma. Whi.	Kol. Smi.	Ma. Whi.	Kol. Smi.	Ma. Whi.	Kol. Smi.	Ma. Whi.	Kol. Smi.
H_{01}	NotObf	P<.00001	P<.00001	P<.00001	P<.00001	P<.00001	P<.00001	0.00419	P<.00001
H_{02}	Obf1			P<.00001	P<.00001	P<.00001	P<.00001	P<.00001	P<.00001
H_{02}	Obf2					P<.00001	P<.00001	0.00709	P<.00001
H_{02}	Obf3							P<.00001	P<.00001

(C)

Table 2. P-value of Hypotheses Testing (Mann-Whithney & Kolmogorov-Smirnov) for f_1 (A), f_2 (B) and f_3 (C)

Hyp.	*f1*	Obf1		Obf2		Obf3		Obf4	
		Ma. Whi.	Kol. Smi.	Ma. Whi.	Kol. Smi.	Ma. Whi.	Kol. Smi.	Ma. Whi.	Kol. Smi.
H_{01}	NotObf	P<.00001	P<.00001	P<.00001	P<.00001	P<.00001	P<.00001	0,00030	P<.00001
H_{02}	Obf1			P<.00001	P<.00001	P<.00001	P<.00001	P<.00001	0.70160
H_{02}	Obf2					P<.00001	P<.00001	P<.00001	P<.00001
H_{02}	Obf3							P<.00001	P<.00001

(A)

Hyp.	*f2*	Obf1		Obf2		Obf3		Obf4	
		Ma. Whi.	Kol. Smi.	Ma. Whi.	Kol. Smi.	Ma. Whi.	Kol. Smi.	Ma. Whi.	Kol. Smi.
H_{01}	NotObf	P<.00001	P<.00001	P<.00001	P<.00001	P<.00001	P<.00001	P<.00001	P<.00001
H_{02}	Obf1			P<.00001	P<.00001	0.04667	0.004667	P<.00001	0.15953
H_{02}	Obf2					P<.00001	P<.00001	P<.00001	P<.00001
H_{02}	Obf3							P<.00001	P<.00001

(B)

Hyp.	*f3*	Obf1		Obf2		Obf3		Obf4	
		Ma. Whi.	Kol. Smi.	Ma. Whi.	Kol. Smi.	Ma. Whi.	Kol. Smi.	Ma. Whi.	Kol. Smi.
H_{01}	NotObf	P<.00001	P<.00001	P<.00001	P<.00001	P<.00001	P<.00001	P<.00001	P<.00001
H_{02}	Obf1			0.72116	0.72116	0.99581	0.99581	P<.00001	P<.00001
H_{02}	Obf2					P<.00001	P<.00001	P<.00001	P<.00001
H_{02}	Obf3							P<.00001	P<.00001

(C)

4.5. Size Effect Analysis

While the above tests allow for checking the presence of significant differences, they do not provide any information about the magnitude of such differences. To this aim, we used the Cohen d effect size, which indicates the magnitude of a main factor treatment effect on the dependent variables. The effect size is considered small for $d \geq 0.2$, medium for $d \geq 0.5$ and large for $d \geq 0.8$ [7]. For independent samples, to be used for unpaired analyses, it is defined as the difference between the means (M1 and M2), divided by the pooled standard deviation: $d = (M1 - M2)/\sigma$. The results are reported in Table 3.

Table 3. Cohen d Coefficient for the Six Samples Corresponding to the Metrics used for Detecting the Obfuscated Code

	Obf1	Obf2	Obf3	Obf4
N-Gram	1.32	3.66	1.17	1.79
Entropy	1.26	2.91	0.95	4.85
Wordsize	0.95	2.47	0.45	0.69
f1	1.28	4.05	1.16	0.60
f2	1.24	4.65	1.18	0.75
f3	1.27	4.28	1.20	0.35

The magnitude of main factor is large for all the cases except for two cases of Obf4 (f_1, which is medium, and f_3, which is small). The effect size is widely recognized in many scientific fields as an indicator of the strength of the relationships between the pairs of samples compared. A large effect size provides a strong guarantee about the reliability of observations for the validity of the experiment conclusions.

5. Conclusion

Code obfuscation is widely used to hide malware, thus it is needed to have mechanisms to detect obfuscated code before establishing whether the code is harmful or not. Much effort has been spent by researchers for defining methods which could de-obfuscate code or recognizing malicious code. At the best knowledge of the authors, very few results have been produced to automatic detecting obfuscated code. This paper investigates the effectiveness of three metrics (n-gram, entropy and wordsize) proposed in literature for detecting obfuscated code.

Three conclusions can be drawn by our experiment:

1. The three metrics are not equally effective: wordsize seems to be successful in all the cases, even if the variability of the sample produced is high in one case and this could affect the detection precision.
2. The effectiveness of the techniques depend on the used obfuscators. There are obfuscation techniques that could be more effective than others, i.e. they successfully camouflage obfuscated code.
3. A new metric, originated by a combination of the three metrics, is more effective than individual metrics and it could be a good candidate for automatically recognizing the obfuscated code.

These metrics should be embedded in web browsers, web content filters and antivirus software for detecting obfuscated code on the flight and signal it to the user, similarly to components that recognize the presence of a pop up in the web page.

The main threat to the validity of our experiment concerns the obfuscators. In this experimentation we used four obfuscators and even if they represent well the populations of code obfuscators available, it could be useful to replicate this experimentation with other obfuscators, in order to enforce the external validity of the experimentation.

As future research, we plan to replicate similar investigations with different obfuscation techniques, as in our experimentation we did not cover all the existing obfuscation strategies. This will help to strengthen the external validity of our findings.

References

[1] P. Akritidis, E. Markatos, M. Polychronakis and K. Anagnostakis, "STRIDE: Polymorphic Sled Detection through Instruction Sequence Analysis", In Proceedings of the 20th IFIP International Information Security Conference (SEC'05), **(2005)**, pp. 375–392.

[2] C. H. Al-Taharwal, H. K. Mao, K. P. Pao, C. F. Wu, H. M. Lee, S. M. Chen and A. B. Jeng, "Obfuscated Malicious JavaScript Detection by Causal Relations Finding", In Proceedings of the 2nd International Conference of Advancements in Computing Technology (ICACT2011), AICIT, **(2011)**, pp. 787-792.

[3] M. Ceccato, M. Di Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano and P. Tonella, "The effectiveness of source code obfuscation: An experimental assessment", In Proceedings of the IEEE 7h International Conference on Program Comprehension (ICPC '09), **(2009)**, pp. 178-187.

[4] S. Chenette, "The ultimate deobfuscator", http://securitylabs.websense.com/ content/Blogs/3198.aspx.

[5] Y. H. Choi, T. G. Kim and S. Choi, "Automatic detection for Javascript Obfuscation Attacks in Web Pages through String Pattern Analysis", International Journal of Security and Its Applications, vol. 4, no. 2, **(2010)**, pp. 13-26.

[6] Y. Choi, T. Kim, S. Choi and C. Lee, "Automatic Detection for JavaScript Obfuscation Attacks in Web Pages through String Pattern Analysis", In Proceedings of the 1st International Mega-Conference on Future Generation Information Technology (FGIT '09), **(2009)**, pp. 160 - 172.
[7] J. Cohen, "Statistical power analysis for the behavioral sciences (2nd ed)", Lawrence Earlbaum Associates, Hillsdale, NJ, **(1988)**.
[8] C. Collberg, C. Thomborson and D. Low, "A taxonomy of obfuscating transformations", Technical Report 148, Dept. of Computer Science, The Univ. of Auckland, **(1997)**.
[9] Computer Security Group at UCSB, "Wepawet", http://wepawet.cs.ucsb.edu/.
[10] M. Cova, C. Kruegel and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious JavaScript code", In Proceedings of he 19th International Conference on World wide web(WWW'10), **(2010)**, pp. 281-290.
[11] M. D. Preda, "Code Obfuscation and Malware Detection by Abstract Interpretation", Ph.D. Thesis, Universit à degli Studi di Verona, Dipartimento di Informatica, TD-02-07, **(2007)**.
[12] S. M. Darwish, S. K. Guirguis and M. S. Zalat, "Stealthy code obfuscation technique for software security", Proc. Of Int'l Conf. of Computer Engineering and Systems (ICCES), **(2010)**, pp. 93- 99.
[13] B. Eich, "SpiderMonkey (JavaScript-C) Engine", http://www.mozilla.org/js/spidermonkey/.
[14] B. Harstein, "jsunpack", http://jsunpack. jeek.org/dec/go/.
[15] http://code.google.com/p/v8/.
[16] J. E. Likarish and I. Jo, "Obfuscated malicious JavaScript detection using classification techniques", In Proceedings of the 4th International Conference on Malicious and Unwanted Software (Malware), **(2009)**, pp. 47-54.
[17] P. Likarish and E. Jung, "A targeted web crawling for building malicious JavaScript collection", In Proceeding of the ACM first international workshop on Data-intensive software management and mining (DSMM '09), **(2009)**, pp. 23–26.
[18] P. Likarish, E. Jung and I. Jo, "Obfuscated Malicious Javascript Detection using Classification Techniques", In Proceedings of the 4th International Conference on Malicious and Unwanted Software, **(2009)**, pp. 47-54.
[19] D. Low, "Protecting Java code via code obfuscation", MagazineCrossroads - Speical issue on robotics Crossroads, vol. 4, Issue 3, **(2008)**, pp. 21-23.
[20] Mozilla Foundation, "Rhino: JavaScript for Java", http://www.mozilla. org/rhino/.
[21] J. Nazario, "Reverse Engineering Malicious JavaScript", In Proceedings of CanSecWest, **(2007)**, http://cansecwest.com/csw07archive.html.
[22] N. Provos, P. Mavrommatis, M. Rajab and F. Monrose, "All Your iFRAMEs Point to Us", In Proceedings of the USENIX Security Symposium, **(2008)**.
[23] The SANS Institute, "Sans top-20 2007 security risks", http://www.sans.org/top20/.
[24] B. Spasic, "Malzilla: Malware Hunting Tool", http://malzilla. sourceforge.net/.
[25] H. P.Van Vliet, "Crema-the java obfuscator", **(1996)**.
[26] www.JavaScriptobfuscator.com.
[27] www.JavaScript-obfuscator.com.
[28] www.malzilla.sourceforge.net.
[29] www.w3facile.com.
[30] Q. Yan, R. H. Deng, Y. Li and T. Li, "On the Potential of Limitation-oriented Malware Detection and Prevention Techniques on Mobile Phones", International Journal of Security and Its Applications, vol. 4, no. 1, **(2010)**, pp. 21-30.