# The Power of Obfuscation Techniques in Malicious JavaScript Code: A Measurement Study

Wei Xu, Fangfang Zhang and Sencun Zhu
Department of Computer Science and Engineering
The Pennsylvania State University, University Park
Email:{wxx104, fuz104, szhu}@cse.psu.edu

## Abstract

*JavaScript based attacks have been reported as the top Internet security threats in recent years. Since most of the Internet users rely on anti-virus software to protect themselves from malicious JavaScript code, attackers exploit JavaScript obfuscation techniques to evade the detection of anti-virus software. To better understand the obfuscation techniques adopted by malicious JavaScript code, we conduct a measurement study. We first categorize observed JavaScript obfuscation techniques. Then we conduct a statistic analysis on the usage of different categories of obfuscation techniques in real-world malicious JavaScript samples. We also study the detection effectiveness of 20 most popular anti-virus software against obfuscation techniques. Based on the results, we analyze the cause of the popularity of obfuscation in malicious JavaScript code; the reason behind the choice of obfuscation techniques and the difference between benign obfuscation and malicious obfuscation. Moreover, we also provide suggestions for designing effective obfuscation detection approaches in future.*

## 1. Introduction

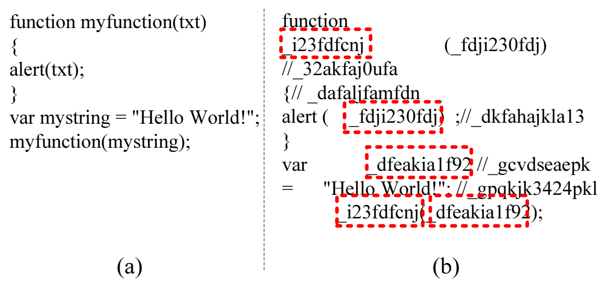
JavaScript based attacks have been reported as the top Internet security threats in recent years [23]. By exploiting numerous vulnerabilities in various Web applications, attackers can launch a wide range of attacks such as cross-site scripting (XSS) [17], cross-site request forgery (CSRF) [19], drive-by downloads [24], etc. To defend against malicious JavaScript code, most Internet users rely on the anti-virus software. Unfortunately, the effectiveness of static signature based anti-virus software is often thwarted by obfuscation techniques. In fact, malicious JavaScript code has been increasingly applying obfuscation techniques to evade the detection of anti-virus software and to hide its malicious intent.

To better understand the usage of obfuscation techniques in malicious JavaScript code, we first survey the obfuscation techniques that have been observed in the wild and categorize them based on the operations performed by these techniques. Then we conduct a measurement study, in which we first analyze the statistics of the usage of different obfuscation techniques by manually examining a set of real world malicious JavaScript samples. Meanwhile, we also study the effectiveness of different obfuscation techniques in evading popular anti-virus software. Based on the results of the measurement study, we are able to provide an in-depth analysis on the cause of the popularity of obfuscation techniques among malicious JavaScript code; the choice of obfuscation techniques among malicious JavaScript code.

We acknowledge that obfuscation techniques are not exclusive to malicious JavaScript code. For example, we find some benign Web pages (e.g., the frontpage of yahoo.com) also use obfuscation to prevent code plagiarism. Therefore, we also discuss the difference between obfuscation techniques adopted by benign JavaScript code and malicious JavaScript code in this paper.

The goal of this work is not to address the problem of detecting obfuscated malicious JavaScript code. However, we believe our results and analysis capture many characteristics of obfuscated malicious JavaScript code and therefore can be leveraged in developing new detection approaches.

The rest of the paper is organized as follows. Section 2 describes a categorization of obfuscation techniques. Section 3 presents the results of our measurement study on the usage of obfuscation techniques and the effectiveness of evading state of the art anti-virus software. Section 4 presents our analysis, followed by



(a)

```
function myfunction(txt)
{
alert(txt);
}
var mystring = "Hello World!";
myfunction(mystring);
```

(b)

```
function
_i23fdfcnj            (_fdji230fdj)
//_32akfaj0ufa
{//_dafalifamfdn
alert ( _fdji230fdj) ;//_dkfahajkla13
}
var        _dfeakia1f92 //_gcvdseaepk
=     "Hello World!"; //_gpqkjk3424pkl
       _i23fdfcnj(_dfeakia1f92);
```

Figure 1. An Example of Randomization Techniques.(a)is the original code and (b) is the obfuscated ocde

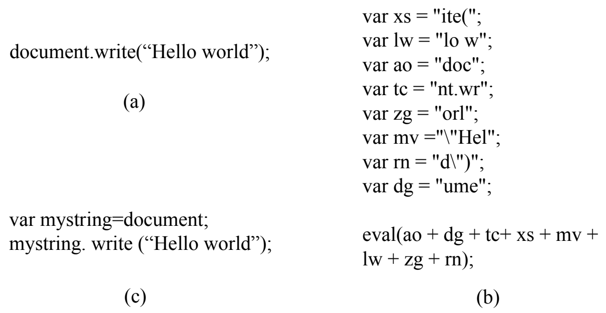
related work in Section 5. We conclude in Section 6.

## 2. Categorizing Observed Obfuscation Techniques in Malicious JavaScript Code

In order to study the usage of obfuscation techniques in a more systematic way, we first classify the observed obfuscation techniques into the following four categories based on the operations performed by them.

**Randomization Obfuscation:** Attackers may randomly insert or change some elements of JavaScript codes without changing the semantics of the codes. Common techniques include *whitespace randomization* [16], *comments randomization* and *variable and function names randomization*.

Whitespace randomization is to randomly insert whitespace characters, including space character, tab, line feed, form feed and carriage return, in JavaScript code. Comments randomization randomly inserts arbitrarily created comments into JavaScript codes. These two take the advantage of the fact that JavaScript interpreters ignore whitespace characters and comments. Variable and function name randomization replaces variable names or function names by randomly created strings with non-obvious meanings. Usually two or more randomization methods are used together to improve the possibility of evading detections. Figure 1 gives a demonstration of these two JavaScript obfuscation techniques. Figure 1(a) is the original code and Figure 1(b) is the obfuscated one. Strings highlighted by red rectangles are randomized variable names and function names. Strings starting with "//" are comments created randomly. Whitespace randomization generates line feed in the end of the first line between "function" and "_i23fdcnj", tab in the second line between "_i23fdcnj" and "(_fdji230fdj)", etc. These two pieces of code have the exactly same semantics, but have different static patterns.



(a)

```
document.write("Hello world");
```

(b)

```
var xs = "ite(";
var lw = "lo w";
var ao = "doc";
var tc = "nt.wr";
var zg = "orl";
var mv ="\"Hel";
var rn = "d\")";
var dg = "ume";

eval(ao + dg + tc+ xs + mv +
lw + zg + rn);
```

(c)

```
var mystring=document;
mystring. write ("Hello world");
```

Figure 2. Examples of string data obfuscation. (a) is the original code; (b) uses string splitting obfuscation; (c) uses keyword substitution

**Data Obfuscation:** Data obfuscation is to convert a variable or a constant into the computational results of one or several variables or constants. Two data obfuscation techniques have been wildly applied to string object. One is *string splitting*. The other one is *keyword substitution*. String splitting is to convert a string into the concatenation of several substrings. String splitting is usually used along with document.write() or eval() functions to execute the concatenated strings in a browser. Attackers could change the order of substrings and assign random variable names to them to make the code even harder to understand. Another obfuscation approach is to use a variable to substitute JavaScript keywords. Examples are shown in Figure 2. Figure 2(a) is the original code[1]. Figure 2(b) uses string splitting obfuscation, where substrings are in a random order to make the code harder to understand and detect. Figure 2(c) uses keyword substitution, where keyword "document" is represented by variable "mystring".

Besides strings, numbers are another object of data obfuscation. For example, $i = 10$ can be rewritten in many ways: $i = 5 * 2$, $i = 11 - 1$ or $i = 1000/100$, etc.

**Encoding Obfuscation:** Normally, there are 3 ways to encode original code. The first way is to convert the code into escaped ASCII characters, unicode or hexadecimal representations. The second method uses customized encoding functions, where attackers usually use an encoding function to create the obfuscated code and attach a decoding function to decode it during execution. Figure 3 shows an example using hexadecimal representation to implement encoding. In Figure 4, we give a simple example of how to obfuscate JavaScript codes by customized encoding and

1. If not specified, we will use this code as the original code in all the following examples in this section

decoding functions: Figure 4(a) is the encoding function, which increases the Unicode of each character by 1, where “document.write(‘Hello world!’)” is encoded into “epdvnfou/xsjuf)(Ifmmp!xpsme(*”; Figure 4(b) is the obfuscated code, which first applies a decoding function, and then executes the decoded instructions. In addition, some standard encryption and decryption methods can be employed to do JavaScript obfuscation. For example, JScript.Encode is a method created by Microsoft to encode JavaScript code. It can be used to protect source code as well as to evade detection.

```
eval("\x64\x6f\x63\x75\x6d\x65\x6e\x74\x2e\x77\x72\x69\x74\
x65\x28\x27\x48\x65\x6c\x6c\x6f\x20\x77\x6f\x72\x6c\x64\x21\
x27\x29");
```

Figure 3. An example of obfuscation using hexadecimal representation.

```
function encode(mystring)
{
var c="";
var i =0;
for(var i=0;i
<mystring.length;i++){
        c = c +
String.fromCharCode(mystring.
charCodeAt(i)+1);
}
return c
}

var c =
encode("document.write('Hello
world!')");
document.write(c);
```

(a)

```
function decode(c)
{
var mystring="";
var i =0;
for(var i=0;i <c.length;i++){
     mystring = mystring +
String.fromCharCode(c.charCodeAt
(i)-1);
}
return mystring;
}

var c ="epdvnfou/
xsjuf)(Ifmmp!xpsme(*";
var mystring = decode(c);

eval(decode(c));
```

(b)

Figure 4. An example of obfuscation using encoding/decoding functions. (a) is the encoding function; (b) is the obfuscated code.

**Logic Structure Obfuscation:** This type of obfuscation technique is to manipulate the execution paths of JavaScript codes by changing the logic structure, without affecting the original semantics. There are two ways to implement logic structure obfuscation. One way is to insert some instructions which are independent of the functionality. The other one is to add or change some conditional branches, such as *if ... else, switch ... case, for, while,* etc. Examples are shown in Figure 5, where Figure 5(a) inserts independent instructions; Figure 5(b) uses additional conditional branches.

```
var i = 111;
i = i+1;
if(i<10)
{
        alart("Warning");
}
document.write('Hello world');
i = i+1;
```

(a)

```
var i =0;
for(i=0;i<=10000;i++)
{
   if(i==1)
   {
      document.write('Hello world');
   }
}
```

(b)

Figure 5. Examples of logic structure obfuscation. (a)inserts independent instructions; (b)uses additional conditional branches.

## 3. Measurement Study

Given the categorization, we study the usage of obfuscation in malicious JavaScript code and the effectiveness of different categories of obfuscation techniques in evading anti-virus software.

### 3.1. Sample Collection and Screening

The malicious JavaScript samples used in our study are collected from VirusTotal [6], which provides a free interface to scan uploaded files using more than 40 state-of-the-art anti-virus software. We selected 1039 malicious HTML samples that have been detected by more than 5 anti-virus software. Since there is no way to only select malicious JavaScript samples, we have to filter out the malicious HTML samples that do not contain any JavaScript code. There are 248(23.9%) such samples, among which 92 do not include any script code and 156 use other scripting languages (e.g., VBScript). The remaining 791(76.1%) samples contain JavaScript codes. Although all 791 samples contain malicious JavaScript code, not all of these samples are detected as malicious because of malicious JavaScript code. For example, an HTML web page that contains only benign JavaScript code can still be malicious because of an embedded malicious URL. In order to obtain the most representative malicious JavaScript samples, we further filter the sample set by two criteria: 1) only keep one sample among samples that are in the same malicious JavaScript family (determined by name); 2) only keeping samples that are reported by more than 15 (out of 20) anti-virus software. We choose the most popular 20 anti-virus software in the market (the selection process will be discussed in Section 3.3) to improve our confidence in the verdict of a sample. Eventually, there are 510 samples left in our sample set.

## 3.2. The Usage of Obfuscation Techniques in Malicious JavaScript Code

Since, to the best of our knowledge, there is no automatic tools or anti-virus software that can detect and categorize JavaScript obfuscation with desired accuracy, we randomly choose 100 samples from our sample set and manually analyze these samples based on the categories we proposed in Section 2.

We find 71% samples use various JavaScript obfuscation techniques. This indicates that obfuscation is a common practice among malicious JavaScript codes to evade detections and to pose an obstacle to code analysis. Among these 71%, 30% of them use at least two types of obfuscation techniques to better hide their malicious purposes. Figure 6 shows the distribution on the number of obfuscation technique categories employed in the samples.

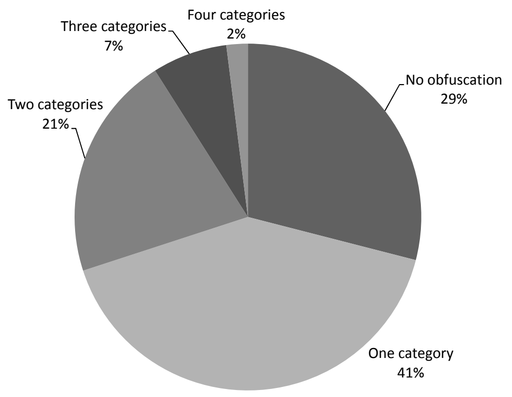


Figure 6. The Distribution on the Number of Obfuscation Technique Categories employed in samples

We also compare the popularity of various obfuscation techniques. The result in Table 1 shows that string computation, e.g. string splitting and keyword substitution, is the most popular method. ASCII/Unicode/Hex code and customized encoding functions are also very common.

## 3.3. The Effectiveness of Evading Anti-virus Software

In this section, we apply different types of obfuscation techniques categorized previously to obfuscate the malicious JavaScript samples in our sample set. After that, we use 20 most popular anti-virus software to scan these obfuscated codes to test the effectiveness of obfuscation techniques in evading anti-virus software. Although 71% of the sample set already adopt various obfuscation techniques, the fact that these samples can be detected by anti-virus software leads to the conclusion that anti-virus software must have generated signatures on the obfuscated malicious code. Therefore, whether these samples contain obfuscated malicious JavaScript code or not, in this test, we will apply another layer of obfuscation to these samples so that we can test the applied obfuscation technique's actual effectiveness of evading anti-virus software.

Since we could not find a widely accepted ranking of anti-virus tools, we choose 20 anti-virus software with the highest rankings on Alexa [1] (in the "Computers/Security" category). We verify our selection by comparing it to some other anti-virus ranking Web sites, e.g., [2] [3] [5] [7]. We noticed that in the top 10 anti-virus software of each ranking list, at least 8 are included in our selection. Therefore, we believe our selection is representative. Table 2 shows the names of these 20 selected anti-virus software.

Before we use the 20 selected anti-virus software to scan further-obfuscated samples, we list the detection rates of these 20 anti-virus software on the samples in our sample set. As shown in Figure 7, the average detection rate is 86.85%.

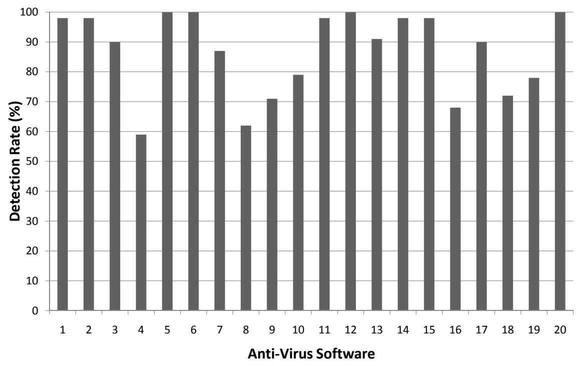


Figure 7. The Detection Rates of 20 Anti-virus Software on Our Sample Set

**3.3.1. Detection against Randomization Obfuscation.** In this section, we obfuscate samples in our data set by randomly adding whitespace and comments, and changing variable and function names to randomly created strings. Figure 8 shows the detection rate of these anti-virus software on the obfuscated samples. The average detection rate is 55.3%, which is much lower than the previous one. Only No.2 and No.20 software demonstrate strong resistance to randomization obfuscation, while all others could be relatively easily bypassed. Because whitespace and comments do

Table 1. The usage of JavaScript Obfuscation Techniques

| Obfuscation Category | | Number |
|---|---|---|
| Randomization Obfuscation | Whitespace Randomization | 3 |
| | Variable and Function Names Randomization | 11 |
| | Comments Randomization | 2 |
| Data Obfuscation | String | 45 |
| | Number | 2 |
| Encoding Obfuscation | ASCII/Unicode/Hex Coding | 32 |
| | Customized Encoding Functions | 23 |
| | Standard Encryption and Decryption | 3 |
| Logic Obfuscation | Insert Irrelevant Instructions | 8 |
| | Additional Conditional Branches | 3 |

Table 2. The Selection of Anti-virus Software

| | |
|---|---|
| 1.a-squared(Emsi Software GmbH) | 11.F-Secure(F-Secure) |
| 2.AntiVir(Avira) | 12.GData(G DATA Software) |
| 3.Avast!Antivirus(ALWIL) | 13.Malware Protection(Microsoft) |
| 4.AVG(AVG Technologies) | 14.Norton Antivirus(Symantec) |
| 5.AVP(Kaspersky Lab) | 15.PCTools(PC Tools) |
| 6.BitDefender(BitDefender GmbH) | 16.Rising(Rising Antivirus) |
| 7.ClamAV (ClamAV) | 17.SAV(Sophos) |
| 8.Comodo(Comodo) | 18.TrendMicro(Trend Micro) |
| 9.DrWeb Doctor(Web,Ltd.) | 19.Vet(CA Inc.) |
| 10.ESET NOD32(Eset Software) | 20.VirusScan(McAfee) |

not change any feature of the codes, we believe that the other 18 anti-virus software rely on static signatures.

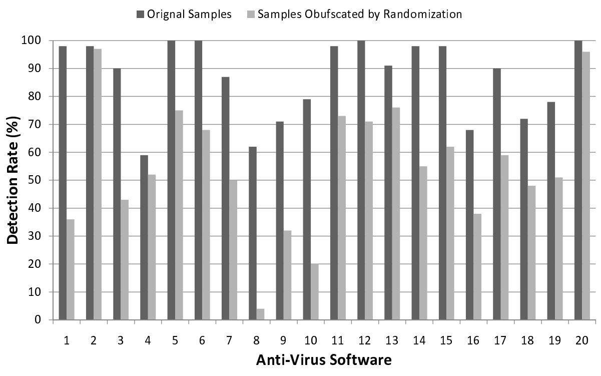


Figure 8. The Detection Rate of 20 Anti-Virus Software on Samples Obfuscated by Randomization

**3.3.2. Detection against Data Obfuscation.** In this section, we obfuscate samples in our data set by using string computation, keyword substitution, and/or number computation. Figure 9 shows the detection rate on obfuscated codes. The average detection rate is 45.7%. The results shows that data obfuscation is more effective to evade detection than randomization.

In one sample, we split a long string into many substrings, and then concatenate them together. All we change is adding ”+” between two adjacent substrings. 16 anti-virus software cannot detect the obfuscated sample as malicious. It indicates that, with a high probability, those anti-virus software use exact matching to compare with the signatures. We also saw another extreme example: there is such an instruction in the sample:

```
document.write(cxw.value.replace(”$PR”,
”&#109;s-its:&#109;html:file://c:\\nosuc
h.mht!http://autoescrowpay.com/chm/xch
m.chm::/launch.html”)
```

We split the second parameter of cxw.value.replace() into 5 substrings, use variables to represent these substrings, and then concatenate them together. Obviously, the semantic of the sample does not change. However, none of the 20 anti-virus software can detect it as malicious. It is also a strong evidence of the fact that these anti-virus software highly relay on signatures. For this sample, the signature is located in the string which we split.

**3.3.3. Detection Rate against Encoding Obfuscation.** We obfuscate the samples using encoding obfuscation, e.g. Hexadecimal encoding. The encoded code is first decoded and then executed by invoke “eval()”. The results show that none of the 20 anti-virus software can detect these encoding obfuscated samples.

**3.3.4. Comparison of Obfuscation Techniques.** Based on our experiment results, we believe that even simple obfuscation methods could effectively evade the detection of anti-virus software. We also believe

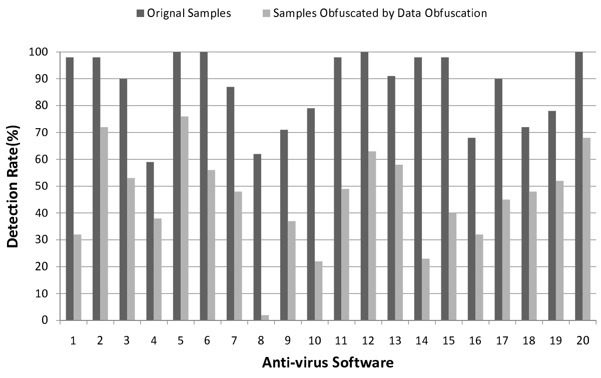


Figure 9. The Detection Rate of 20 Anti-Virus Software on Samples Obfuscated by Data Obfuscation

that most popular anti-virus software use signature-based detection scheme. Some of them even use exact matching.

## 4. Analysis and Discussion

### 4.1. The Cause of the Popularity of Obfuscation

The popularity of obfuscation among malicious JavaScript code is caused by the following reasons. First, signature based detection systems, e.g., anti-virus software, can be effectively evaded by obfuscation. As we have demonstrated, by applying encoding obfuscation, all the obfuscated samples can successfully evade the detection of any state-of-the-art anti-virus software in our study.

Second and more importantly, we discover that the dynamic features of JavaScript language such as dynamic code generation and run-time evaluation often facilitate the creation of obfuscation routines. Since these two features provide a means of transforming text to code in JavaScript, at a high level, any string manipulation process can be combined with dynamic generation/run-time evaluation function (e.g., "document.write()", "document.writeln()", "eval()", "window.setTimeout()", "window.setInterval()") to generate an obfuscation routine (e.g., [4]). Therefore, we believe the examination of the arguments to these functions can be leveraged in the detection of obfuscated malicious JavaScript code.

Third, many of the obfuscation techniques can be stacked together to generate a multi-level obfuscation scheme. We noticed that this feature makes the reuse of obfuscated malicious code more easily since attackers do not need to deobfuscate an obfuscated code before applying another obfuscation technique. Therefore, when an obfuscated malicious code is detected by anti-virus software, attackers can simply wrap the obfuscated code with another layer of obfuscation to further evade detection.

Fourth, the improvement in the performance of JavaScript engine in current web browsers further helps to promote the popularity of obfuscation as well. We noticed that the execution time difference between un-obfuscated code and obfuscated code is almost negligible. Therefore, attackers no longer have to concern that the time difference will cause user's suspect of any malicious activity. Moreover, with the improvement of performance, we also observe that nearly half of the obfuscated samples actually apply multiple levels of obfuscation to better hide their malicious intent.

### 4.2. The Choice of Obfuscation Technique by Malicious JavaScript Code

As the results illustrated, the top two most popular obfuscation technique is data obfuscation (e.g., string splitting) and encoding obfuscation (e.g., ASCII/Unicode/Hex encodeing). This is because:1) both techniques can be effortlessly applied to any JavaScript code without considering the logic of the code; 2) these two techniques can reduce the detection rate by around 40% and 100% respectively.

However, data obfuscation can not evade the detection of anti-virus software as effectively as encoding obfuscation. Therefore, we believe an effective obfuscation detection approach should focus on encoding obfuscation.

### 4.3. Benign Obfuscation vs Malicious Obfuscation

Both benign and malicious JavaScript code have been observed adopting obfuscation techniques; hence, obfuscation does not imply maliciousness. However, their purposes of obfuscation are different. Benign JavaScript code mainly leverages obfuscation to protect code privacy or intellectual property. This purpose requires obfuscated code to be human unreadable. Malicious JavaScript code exploits obfuscation to hide its malicious intent; therefore, the obfuscated code aims to evade automatic static inspection.

## 5. Related Work

A number of studies have discussed the usage of JavaScript obfuscation. Kolisar [21] provided a survey of current JavaScript obfuscation techniques, where he only briefly enumerated several simple techniques without considering the systematic classification. Feinstein *et al.* [16] discussed JavaScript obfuscation techniques based on their complexities, but they did not study the usage of these techniques. Chellapilla *et al.* [9] indicated that obfuscation techniques are prevalent among JavaScript redirection spam pages. Different from these studies, our work provides a comprehensive measurement study on the usage of JavaScript Obfuscation based on a systematic classification.

In the area of JavaScript obfuscation detection, several approaches have been proposed. In [20], which is a follow-up of Zozzle [15], Kaplan *et al.* leverages an AST based static classifier. NoFus detects if a piece of JavaScript code has been obfuscated for any purpose, and it also distinguishes benign obfuscated JavaScript and malicious JavaScript. In [11] Choi *et al.* proposed an automatic approach to detecting JavaScript obfuscation based on metrics such as N-gram and entropy. Büscher *et al.* [8] propose Monkey-Wrench, a web-honeyclient that performs both static and dynamic analysis on web pages in an emulated browser environment. Likarish *et al.* proposed a classification-based scheme to detect obfuscated malicious JavaScript codes [22], but there is no justification on their selection of features. There are also some automated JavaScript analyzing tools, such as Caffeine Monkey [16], Wepawet [14], Jsunpack [18], and Toor-ConX [10]. These tools are more powerful, but they do not identify the obfuscation techniques being used. Moreover, since these tools apply dynamic analysis, for example, by examining the runtime behavior of obfuscated codes in a customized JavaScript engine, they suffer more from performance penalties when compared with static analysis.

In addition, none of these papers or projects measure the detection effectiveness of popular commercial anti-virus products against JavaScript Obfuscation, whereas we believe such a study is very meaningful for us to understand the incapabilities of these commercial products.

There are also several works focus on the classification of obfuscation in other programming languages. Collberg *et al.* [13] proposed three categories of control-flow obfuscation techniques. Their work focuses on Java code while this work focuses on JavaScript code. Christodorescu *et al.* [12] tested the resilience of anti-virus software to four obfuscation techniques on VB malware. Their results also demonstrated that commercial anti-vitus software are not resilient to common obfuscation. The difference between [12] and this work is that we also discuss the cause of popularity of JavaScript obfuscation; the reason behind the choice of obfuscation techniques and the difference between benign and malicious obfuscation.

## 6. Conclusion and Future Work

In this paper, we conduct a measurement study on the usage of obfuscation techniques in malicious JavaScript code. We find that JavaScript obfuscation is a common practice among malicious JavaScript code in order to evade detections. Moreover, some malicious code employs multiple levels of obfuscation to further complicate the analysis and to better hide their malicious purposes. The results demonstrate that all popular anti-virus products can be effectively evaded by various obfuscation techniques.

From this measurement study, we realize that JavaScript obfuscation techniques are dangerous threats to Web security because of the simplicity of applying such techniques on existing malicious codes and the lack of efficient and effective detection approaches. Therefore, we believe there is a great need for a scheme that can achieve both *high accuracy* and *high performance* in detecting obfuscated malicious JavaScript. However, as discussed in related work, signature-based anti-virus software suffering from low detection rate, while those tools heavily relying on dynamic analysis may suffer from performance penalties. Therefore, we envision an approach that can combine both dynamic analysis and static analysis to yield desired detection performance, and we hope our work will provide some motivation for developing such a scheme in the future.

## 7. Acknowledgement

This work was supported in part by the NS-CTA grant from the Army Research Laboratory and NSF-0643906. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of ARL or NSF.

## References

[1] "Alexa." [Online]. Available: http://www.alexa.com. (last accessed at March 18, 2010)

[2] "And the best antivirus is..." [Online]. Available: http://cybernetnews.com/and-the-best-antivirus-is(last accessed at March 18, 2010)

[3] "Antivirus software review 2010." [Online]. Available: http://anti-virus-software-review.toptenreviews.com/(last accessed at March 18, 2010)

[4] "Online JavaScript Obfuscator," http://www.daftlogic.com/projects-online-javascript-obfuscator.htm.

[5] "Top ten antivirus with top 10 best antivirus." [Online]. Available: http://www.toptenservices.net/2009/03/24/top-ten-antivirus-programs/(last accessed at March 18, 2010)

[6] "Virustotal." [Online]. Available: http://www.virustotal.com. (last accessed at March 18, 2010)

[7] "Worlds widely used antivirus softwares." [Online]. Available: http://infozblog.com/worlds-widely-used-antivirus/(last accessed at March 18, 2010)

[8] A. Büscher, M. Meier, and R. Benzmüller, "Throwing a monkeywrench into web attackers plans," in *Proceedings of the 11th IFIP TC 6/TC 11 international conference on Communications and Multimedia Security*, ser. CMS'10, 2010, pp. 28–39.

[9] K. Chellapilla and A. Maykov, "A taxonomy of javascript redirection spam," in *The International Workshop on Adversarial Information Retrieval on the Web (AIRWeb)*, May 2007.

[10] S. Chenette, "Toorconx." [Online]. Available: http://securitylabs.websense.com/content/Blogs/3198.aspx (last accessed at March 18, 2010)

[11] Y. Choi, T. Kim, S. Choi, and C. Lee, "Automatic detection for javascript obfuscation attacks in web pages through string pattern analysis," in *Future Generation Information Technology(FGIT)*. Springer-Verlag Berlin Heidelberg, May 2009, pp. 160–172.

[12] M. Christodorescu and S. Jha, "Testing malware detectors," in *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, ser. ISSTA '04, 2004, pp. 34–44.

[13] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '98, 1998, pp. 184–196.

[14] U. computer security lab, "Wepawet." [Online]. Available: http://wepawet.iseclab.org/index.php(last accessed at March 18, 2010)

[15] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert, "Zozzle: Fast and precise in-browser javascript malware detection," in *Proceedings of the 20th conference on USENIX security symposium*. USENIX Association, 2011.

[16] B. Feinstein and D. Peck, "Caffenie monkey: automated collection detection and analysis of malicious javascript," in *Black Hat*, 2007.

[17] J. Grossman, R. Hansen, P. D.Petkov, A. Rager, and S. Fogie, *XSS Attacks: Cross Site Scripting Exploits and Defense*, 2007.

[18] B. Harstein, "Jsunpack." [Online]. Available: http://jsunpack.jeek.org/dec/go(last accessed at March 18, 2010)

[19] N. Jovanovic, E. Kirda, and C. Kruegel, "Preventing cross site request forgery attacks," in *Second IEEE Communications Society/CreateNet International Conference on Security and Privacy in Communication Networks(Securecomm)*, September 2006, pp. 1–10.

[20] S. Kaplan, B. Livshits, B. Zorn, C. Siefert, and C. Curtsinger, ""nofus: Automatically detecting" + string.fromcharcode(32) +"obfuscated ".tolowercase() + "javascript code"," Microsoft Research, Tech. Rep., 2011.

[21] Kolisar, "Whitespace: A different approach to javascript obfuscation," in *DEFCON 16*, August 2008.

[22] P. Likarish, E. E. Jung, and I. Jo, "Obfuscated malicious javascript detection using classification techniques," in *IEEE International Conference on Malicious and Unwanted Software*, October 2009.

[23] N. J. Percoco, "Global security report 2010 analysis of investigations and penetration tests," SpiderLabs, Tech. Rep., 2010.

[24] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose, "All your iframes point to us." in *USENIX Security Symposium*, 2008, pp. 1–15.