2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

Obfuscated VBA Macro Detection Using Machine Learning

Sangwoo Kim, Seokmyung Hong, Jaesang Oh and Heejo Lee*
Korea University
Seoul, Republic of Korea
Email: { sw_kim, canasta, jaesangoh, heejo } @korea.ac.kr

***Abstract*—Malware using document files as an attack vector has continued to increase and now constitutes a large portion of phishing attacks. To avoid anti-virus detection, malware writers usually implement obfuscation techniques in their source code. Although obfuscation is related to malicious code detection, little research has been conducted on obfuscation with regards to Visual Basic for Applications (VBA) macros.**

In this paper, we summarize the obfuscation techniques and propose an obfuscated macro code detection method using five machine learning classifiers. To train these classifiers, our proposed method uses 15 discriminant static features, taking into account the characteristics of the VBA macros. We evaluated our approach using a real-world dataset of obfuscated and non-obfuscated VBA macros extracted from Microsoft Office document files. The experimental results demonstrate that our detection approach achieved a F_2 score improvement of greater than 23% compared to those of related studies.

I. Introduction

Attacks using macros have become a constant threat since the "*Concept*", a wide-spread macro virus written in Visual Basic for Applications (VBA), which appeared in 1995 [1]. Macro malware was a major threat from the late 1990s to the early 2000s, but it had declined since the security mechanism of Microsoft Office was enhanced in 2000 [2], [3]. However, according to the statistics and security news of Anti-Virus (AV) companies, attacks using VBA macros have been increasing again since the second half of 2014 [4], [5]. Since the release of Microsoft Office 2000, the execution of VBA macros was disabled by default, but attackers began to deploy simple social engineering techniques that lure users into enabling the execution of macros.

The threat reports of AV companies also confirmed the comeback of script-based malware such as VBA macro malware. According to the report released by Symantec in 2016, MS Office document file formats dominated the email attachments (73.2%), even more than executable files [6]. Furthermore, a recent Kaspersky threat report demonstrates that the Microsoft Office Word VBA macro-based attacks are included in the top 10 malware families [7]. The latest McAfee security report, published in September 2017, also covers the trends of script-based malware and reports a malware type which includes PowerShell command inside of VBA macro [8].

* Heejo Lee is the corresponding author.

As mentioned above, the security reports of AV vendors have shown that script-based attacks are on the rise and can be dangerous. The most frequently mentioned scripting languages that can be used in malicious code are JavaScript, Visual Basic Script, PHP, and Powershell. Among these, VBA macro malware, which is an attack related to MS Office documents, should not be ignored. Owing to the fact that MS Office document files are used by a large number of companies and institutions, malware which leverages MS Office documents as an attack vector can have a large impact. Attacks related to VBA macros are usually considered less suspicious than the executable files because most people are familiar with the MS Office document files, e.g., .docx or .pptx. In result, this negligence leads to the proliferation of ongoing VBA macro attacks.

A primary quality that a successful cyber-attack must have is the ability to bypass AVs. One of the most effective strategies to bypass AVs is obfuscation, which is the intentional obscuring of code by making it difficult to understand. In many script-based malware, obfuscation techniques are fairly common, and it is generally known that obfuscation works well against AVs. There have been malicious JavaScript detection studies which categorized obfuscation techniques into four types and investigated how the detection rate changed when they were applied [9], [10]. The studies demonstrated that obfuscation techniques are effective in avoiding the AV detection.

Currently, many obfuscated VBA macro attacks are underway, but there are still few studies on obfuscated VBA macro detection. Most document malware detection researches focused on vulnerability or shellcode detection [11]–[14].

Only recently has it appeared in several studies under the name of "Downloader" or "Macro malware". Mimura et al. [15] conducted a study to extract the Remote Access Trojan (RAT) in malicious documents files used in Advanced Persistent Threat (APT) attacks from 2009 to 2015. They classified the collected document malware as "Downloader" and "Dropper". "Downloader" uses VBA macro, and "Dropper" includes executable files in itself. However, the focus of the study was on the "Dropper", rather than the 'Downloader".

We have observed that the rate of VBA macro use in APT attacks has been drastically increasing since 2014, and our proposed method targets the missing area that has not been studied by the referenced research. There are few studies that leverage machine learning to detect malicious MS Office documents.

Cohen and Nissim et al. [16], [17] proposed a method to detect malicious docx files with structural features by using active learning that emphasizes the updatability of a detection model. It provided a 94.44% true positive detection rate by leveraging the hierarchical nature of docx files. It presented the most prominent 11 features, including 8 structural path related to the existence of VBA macros.

Conversely, research on detecting attacks related to PDF documents have been widely carried out. VBA macros in MS Office files and JavaScript in PDF documents share similar characteristics. We can detect the obfuscation techniques in the JavaScript of the PDF files, and there are many studies on the detection of obfuscated malicious JavaScript. Given that it is also a scripting language, one may think that we can apply JavaScript research to VBA macro detection, but it has never been demonstrated how it would work. There are many similarities due to the “scripting language in the document”, but the language itself is different, hence the obfuscated code is very different. For instance, there is a minification technique in JavaScript. Although minification can reduce code size by deleting linefeed, it often appears in malicious script code to avoid malware detection. This technique is only applicable to JavaScript, not VBA macros. Owing to the differences between JavaScript and VBA, independent research focusing on VBA macros should be conducted.

In this paper, we propose a method to detect obfuscated VBA macros in MS Office documents by using machine learning classifiers. First, we investigated the VBA macros that were actually used as malicious code, and classified the VBA obfuscation techniques into four categories by referring to related research. In our experiment, we evaluated the performance of our proposed obfuscation detection method which leverages machine learning. 773 malicious and 1,764 benign MS Office files were collected and we conducted an experiment with 4,212 VBA macro extracted from the collected files. All VBA macros were manually labeled as either obfuscated or normal. By performing a manual scan on large, real-world samples, we demonstrated how many malicious and benign samples were obfuscated. From this labeled dataset, we extracted 15 discriminant static features that reflect the characteristics of the VBA macros and applied them to five different classifiers, and compared the results with those of related studies. As a result, we obtained a 23% improvement in F_2 score in our comparative experiment.

The contributions of this paper are as follows.

- As the first obfuscation detection study applied to VBA macro, we have summarized the types of obfuscation techniques and we have shown the extent of obfuscation applied to real-world VBA macros.
- We presented 15 discriminant static features and, tested them using five different classifiers. The results of the comparison with related research show that the performance was improved by 23%.

The rest of this paper is organized as follows. Section II summarizes related studies concerning detection of document malware. Section III provides the simple explanation about VBA macro, and categorization of obfuscation techniques. In Section IV, we propose our obfuscated VBA macro detection approach with experiment setup. Section V, we evaluate the classification performance of proposed detection approach. Finally, Section VI and VII include discussion and conclusion of this paper.

II. Related work

Attacks using VBA macros continue to increase. Moreover, over 98% of malicious VBA macros are obfuscated according to our manual inspection on a collected sample set (as detailed in Section IV.B). However, there is a scarcity of studies on the detection of obfuscation on VBA macros. Given that attacks using VBA macros have only just begun to increase, most research is focused on vulnerability or shellcode detection [11]–[15] rather than on the detection of VBA macros. The following are the studies that can be applied to attacks using VBA macros.

A. Malicious VBA macro detection

Until now, a few studies have been proposed and most of them are based on a machine learning approach. Cohen et al. [16] conducted research on malware detection for XML-based documents. This study uses the hierarchical nature of Office Open XML (OOXML) as a key feature of machine learning to detect MS Office document malware. It recognized the risks that could be posed by document files, and well-organized the types of possible attacks which could result from them. In their experiment, nine different classification algorithms were used and Random Forest classifiers demonstrated the best results among them. In addition, they proved the effectiveness of their proposed method by comparing the detection results to those of several AVs. This research using the idea of structural feature has proven to be effective when dealing with OOXML file types such as .docx, .docm, or .xlsx. However, the majority of VBA macro malware are .doc or .xls, which are not OOXML file types [6].

Subsequently, Nissim et al. [17] added Active Learning to the SFEM method in 2017. Active Learning methods are designed to assist the analytical efforts of experts; it led to a 95.5% reduction of labeling efforts. However, their proposed mechanism is limited to docx files, which is narrower than OOXML files.

Gaustad [18] presented a research on malicious VBA macro detection in 2017. This study used a Random Forest classifier of the ensemble learning with over a thousand static features to detect malicious documents. However, given that its detection was performed with the static features of malicious VBA macro codes, it is difficult to identify how obfuscation techniques were considered in the detection process.

B. Malicious JavaScript code detection

JavaScript is one of the most popular scripting languages. JavaScript-based attacks are also taking place in PDFs, and have similarities with VBA in that both threats utilize scripting language in document formats. By retrieving research on malicious JavaScript detection, we are able to explore appropriate ways to counteract VBA macro malware.

While malicious VBA macro detections in MS Office documents mainly consist of machine learning methods, there are a larger variety of approaches to detecting malicious JavaScript in PDFs. Moreover, a number of research have been emphasizing on analyzing obfuscations, some focusing on restoring the obfuscated code to original code by de-obfuscating it. In the subsections below, we will introduce the representative studies that have been researched to detect malicious JavaScript.

Static analysis approach: In malware detection, static analysis has advantages over dynamic analysis in terms of cost for inspection, because it generally guarantees a lightweight inspection. Choi et al. [19] proposed a method to detect JavaScript obfuscation that leveraged the lexical characteristics of obfuscated strings. Detection was performed by using an N-gram distribution, entropy, the string length for all the strings used, and the parameters of the dangerous function. Xu et al. [20] analyzed the decoding process of obfuscated code to detect obfuscation. Their key idea was that obfuscated malicious JavaScript code has to be de-obfuscated before it executes its malicious actions. They identified the function calls that are related with obfuscated malicious JavaScript code.

Dynamic analysis approach: Liu et al. [21] proposed a method to detect malicious JavaScript through document instrumentation. This method inserts monitoring code into a PDF, so that the inspector knows the context of the runtime behaviors. Kim et al. [22] proposed J-force, which is a forced execution engine for JavaScript. J-force was introduced to detect suspicious hidden behavior, and it achieved a 95% code coverage on real-world JavaScript samples. Furthermore, there is a study focusing on the de-obfuscation of malicious JavaScript, JSDES [23]. It is an automated system for de-obfuscation and analysis of malicious JavaScript code. This study conducted an extensive survey on the available JavaScript obfuscation techniques and their usage in malicious code.

Machine learning approach: Likarish et al. [24] proposed a method based on the Support Vector Machine (SVM) and a decision tree to detect malicious JavaScript in web pages. They proposed a frequency of 50 keywords and 15 properties as detection features that indicate human-readable characteristics. Jodavi et al. [25] used one-class SVM classifiers to detect obfuscation. In training, they pruned the classifier ensemble using a novel binary Particle Swarm Optimization (PSO) algorithm to find a near-optimal sub-ensemble. Aebersold et al. [26] tested the machine learning approach to detect obfuscated JavaScript in 2016. This study trained four different classifiers and evaluated them with real-world PDF files. Their approach and proposed features scored promised results on a benign dataset, but had a 60.6% recall score on a malicious dataset.

```
Sub StartCalculator()
  Dim Program As String
  Dim TaskID As Double
  On Error Resume Next
  Program = "calc.exe"

  'Run calculator program using Shell()
  TaskID = Shell(Program, 1)
  If Err <> 0 Then
  MsgBox "Can't start " & Program
  End If
End Sub
```

(a) A macro for running the program “calc.exe” in Windows

```
Sub SendEmail()
  Dim OutlookApp As Object
  Dim MItem As Object

  'Create Outlook object using CreateObject()
  Set OutlookApp = CreateObject("Outlook.Application")

  For Each cell in
       Columns("a").Cells.SpecialCells(xlCellTypeConstants)
    email_ = cell.Value
    subject_ = cell.Offset(0, 1).Value
    body_ = cell.Offset(0, 2).Value

  'Create Mail Item and send it via Outlook object
  Set MItem = OutlookApp.CreateItem(0)
  With MItem
    .To = email_
    .Subject = subject_
    .Body = body_
    .Display
  End With
  Next
End Sub
```

(b) A macro for sending an email via Outlook

Fig. 1: VBA macro code sample

The users of the host applications are able to leverage the VBA language to write script that access to the functionalities of host applications.

III. Background

A. *Visual Basic for Applications*

Visual Basic for Applications (VBA) is a scripting language that is implemented within host applications, such as Microsoft Office Word or Excel [27]. The advantage of VBA is its ability to automatically and repeatedly use various functions of the host application and system. Figure 1 displays a sample macro code that interacts with a system. Figure 1(a) shows the macro code for executing a program of a system via the VBA function *Shell*(). With several lines of code, any program in a computer can be executed. As shown in Figure 1(b), VBA can be used to send emails in Excel via an Outlook object. Through VBA, users can perform a variety of tasks.

The expandability of VBA is convenient for users, but it can also become an opportunity for attackers. Attackers can accomplish almost every action that can be used for malicious behavior, such as downloading or executing, via a VBA macro. Figure 1 represents the sample code of functions

TABLE I: Type of obfuscation techniques

#	Type	Method
O1	Random obfuscation	Randomize name
O2	Split obfuscation	Split strings
O3	Encoding obfuscation	Encode strings
O4	Logic obfuscation	Insert and reorder code

```
'Procedure name is changed to "ueiwjfdjkfdsv"
Sub ueiwjfdjkfdsv()
  'Variable name is changed to "yruehdjdnnz"
  Dim yruehdjdnnz As Integer
  yruehdjdnnz = 2
  Do While yruehdjdnnz < 45
    DoEvents: yruehdjdnnz = yruehdjdnnz + 1
End Sub
```

Fig. 2: An example of Random obfuscation

that are triggered by users – however attackers prefer to take advantage of functions triggered upon opening a document, such as *workbook_open*() or *document_open*(). Furthermore, by using simple social engineering techniques which lure users to enable macros, attackers are able to bypass MS Office's security mechanism.

B. Obfuscation Techniques in VBA

The goal of this study is to detect obfuscation with the textual characteristics of obfuscated macro code. For more effective detection, we classify obfuscation techniques into four types by target and method of obfuscation based on the studies by Collberg et al. [28] and Xu et al. [9]: *1) Random obfuscation*, *2) Split obfuscation*, *3) Encoding obfuscation*, and *4) Logic obfuscation*. Each obfuscation type has different syntactic structure and different uses of functions and operators. Therefore, we can use the unique characteristics of each type to detect obfuscation. Table I provides a summary of each obfuscation type.

The obfuscation techniques affect the manual code inspection of human experts. Whether it be a signature-based AV or machine learning based AV, in order to judge the maliciousness of code, it must be predetermined by human experts. These obfuscation techniques are applied to decelerate the time of analysis, which in turn, delays the countermeasures after detection. Although each obfuscation technique is quite simple, when used in combination, they render the code visually indecipherable. In addition, attackers use obfuscation tools to create many variants of malware with different hash values. In the following subsections, the explanation of each obfuscation technique and our machine learning features to detect these techniques will be provided with example code.

1) O1 Random Obfuscation: Random obfuscation is a type of obfuscation that changes the identifiers of VBA macro code. Identifiers are the names of variables and procedures that

```
Public Const pzonda = "a"
Public Const pzonde = "e"
Public Const pzondP = "P"

'Parameter "wScript.shell" is divided
CreateObject("WScript.Sh" + pzonde + "ll")
'Parameter "Process" is divided
.Environment(pzondP + "" + "roc" + pzonde + "ss")
```

Fig. 3: An example of Split obfuscation

are used in VBA macro code. Random obfuscation makes it difficult to analyze the flow from variables and function calls by changing the identifiers to random strings.

Figure 2 shows an example of random obfuscation. The names of the sub procedure and the variables are changed to random meaningless strings such as *ueiwjfdjkfdsv*, *yruehdjdnnz*. This change to random strings makes it difficult for humans to understand the actual operation of the macro code.

The identifying feature of this random obfuscation is in the naming of the identifiers. Therefore, using Entropy, a measure of the disorder of the characters of the identifiers, can be one way of detecting the characteristics of this obfuscation. Related studies already leverage the entropy of the entire code as one feature to detect malicious scripts [18], [26]. In addition to this, given that random obfuscation is applied to identifiers, it is also possible to use the variance or mean value of length of identifiers as one feature of obfuscation detection.

Transform of Random Obfuscation

Sub function()	→	Sub uoweghklsdfdw()
Dim variable	→	Dim io3u9nlkq8lqk

2) O2 Split Obfuscation: Split obfuscation usually performs obfuscation by dividing parameter data. The morphological changes that occur in the process of partitioning data have proven to be very effective in avoiding signature-based AVs [9]. As the data is partitioned, it has a form that is different from the detection signature hence, it is not flagged by the detection technique. However, when the macro is executed, the parameter value transferred to the function is the same, so the macro can successfully execute its malicious action. Figure 3 displays an example of macro code with split obfuscation. This conversion does not change the actual behavior of the code, but it avoids the detection of the use of "wScript.shell" or "Process" as the signature for malware detection.

Functions such as *Shell*() and *URLDownloadToFile*() are frequently used for attacks in malicious VBA macros, but legitimate users can also use them in benign VBA macros for normal programs. Therefore, in order to determine whether a VBA macro is obfuscated or not, it is necessary to verify not only the functions it uses, but also the input parameters of the functions. Split obfuscation obstructs the detection of malicious code by modifying parameter values.

```
'Parameter "savetofile" is changed to "savteRKtofilteRK"
Replace("savteRKtofilteRK", "teRK", "e")
```

(a) Obfuscation using built-in function *Replace*()

```
'Each character of URL is changed to number
urlAr = Array(1878, 1890, 1890, 1886, 1832, 1832, 1821,
     1886, 1871, 1890, 1878, 1875, 1884, 1888, 1895, 1879,
     1882, 1891, 1883, 1879, 1884, 1871, 1873, 1879, 1885,
     1884, 1820, 1879, 1830, 1820, 1873, 1885, 1883, 1821,
     1829, 1828, 1876, 1828, 1874, 1827, 1821, 1827, 1826,
     1889, 1874, 1876, 1877, 1829, 1878, 1830, 1880, 1820,
     1875, 1894, 1875)
urlstr = DecodeArray(urlAr)
```

(b) Obfuscation using user-defined function *DecodeArray*()

Fig. 4: An example of Encoding obfuscation

In obfuscated macro code, in order to use the split data, it is essential to combine it. The combination of data is done using the join operators '&' and '+', as shown in Figure 3. The join operators are used in normal macros, but more often in obfuscated macros. Thus, an excessive appearance of these characters can be selected as one of the features to detect obfuscation. In addition to this, given that it also increases the number and length of string variables, we can also leverage it as a feature.

Transform of Split Obfuscation

"String" → "St" & "r" & "in" & "g"

3) O3 Encoding Obfuscation: Encoding obfuscation performs obfuscation by modifying function parameters like split obfuscation. Modification is performed by converting parameter data using reversible algorithms such as Base64 or Shift. Three types of methods are used in encoding obfuscation: 1) built-in VBA functions, 2) character encoding, and 3) user-defined functions.

The first type of encoding obfuscation uses the built-in functions of VBA such as *Replace*(), *Right*(), or *Left*(). Figure 4(a) shows an obfuscation using *Replace*() which is basically supported by VBA. As shown in the figure, the parameter "savetofile" is saved as "savteRKtofilteRK" which replaces "e" to "teRK". It prevents macros from being detected by the keyword "savetofile". The second type of encoding obfuscation changes the character encoding by the use of VBA functions such as *Asc*(), *Hex*(), *Chr*(). These functions change characters to the number of the ASCII code and vice versa. The last type of encoding obfuscation uses conversion algorithms that are manually defined by users, for example, 4(b). Many algorithms are used with simple bitwise operations, such as shift or xor, or complex encryptions, such as Base64.

The functions used for encoding obfuscation are used in non-obfuscated macros as well, but there is a large gap in the frequency of their appearance. This is because attackers encode as many strings as possible to prevent AVs from finding keywords. In the case of "Downloader [15]" which downloads and executes a malicious executable, the URL, path and related strings are all encoded by use of the aforementioned functions. Hence, we can leverage the appearance frequency of encoding functions as a feature to detect this type of obfuscation.

Transform of Encoding Obfuscation

"A"	→	Ord(65)
"String"	→	Replace("Stripe","pe","ng"))
"String"	→	decodeBase64("U3RyaW5n")

4) O4 Logic Obfuscation: Logic obfuscation changes the execution flow of macro code. It complicates the code and makes analysis difficult. This technique is done by declaring unused variables or using redundant function calls. It is not difficult to increase the code size by inserting dummy codes, and it is already being used by a public VBA macro obfuscation tool [29]. If the size of the code that needs to be analyzed increases 100 times by deliberately inserting redundant dummy code, it means that the time it takes for the code analyst to troubleshoot the obfuscated code will be increased by the considerable amount.

Although the logic obfuscation affects the code analysis, it often results in a significant change in code size. It also changes several characteristics of code such as the number of functions and declared variables, function parameters, string data, etc. Therefore, logic obfuscation has no effect on the detection rate in our obfuscation detection study using static features. Rather, if the characteristics of logic obfuscation are well-summarized, we can leverage them as features to detect obfuscation. In Section IV, 15 discriminant static features which reflect the above-mentioned characteristics of the obfuscation techniques will be introduced.

IV. Detecting Obfuscation with a Machine Learning Approach

The obfuscation techniques in VBA macros are explained in Section III. To detect aforementioned obfuscation techniques, we propose a method based on classification algorithm through supervised machine learning. Although machine learning based detection method requires several prerequisites such as sufficient data collection, training set labeling, and feature selection process, it nevertheless has several advantages over alternative techniques. Unlike machine learning, static analyses, such as signature or pattern based detection method, have limitations when counteracting to unknown malware; dynamic analysis has a heavy overhead. On the other hand, machine learning approach has been applied in numerous areas of the computer science field including anomaly detection, and has guaranteed and acceptable run time. If the prerequisites are satisfied, machine learning method can overcome the shortcomings of the above-mentioned approaches and promising performance can be expected.

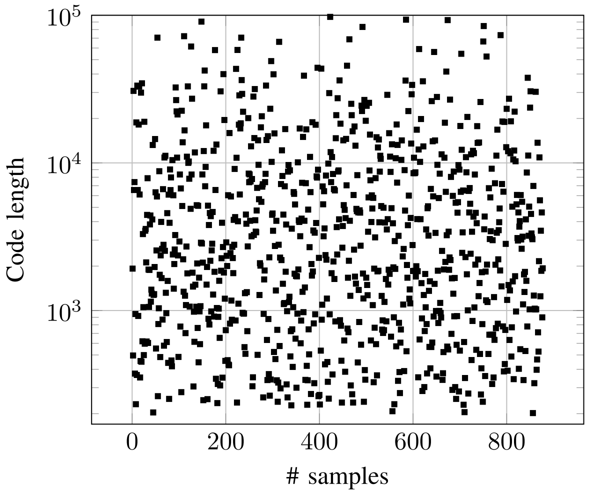
This section provides an overview of our experiment process. It consists of 1) Data collection, 2) Preprocessing, 3) Feature extraction & selection, and 4) Classification using machine learning classifiers. To thoroughly evaluate the performance of our proposed machine learning method, we first explain how we collected the samples and preprocess them. After that, the entire process of extracting and selecting features to effectively detect the obfuscation techniques summarized in Section III will be described. Finally, the explanation of the machine learning classifiers will follow.

A. Data collection

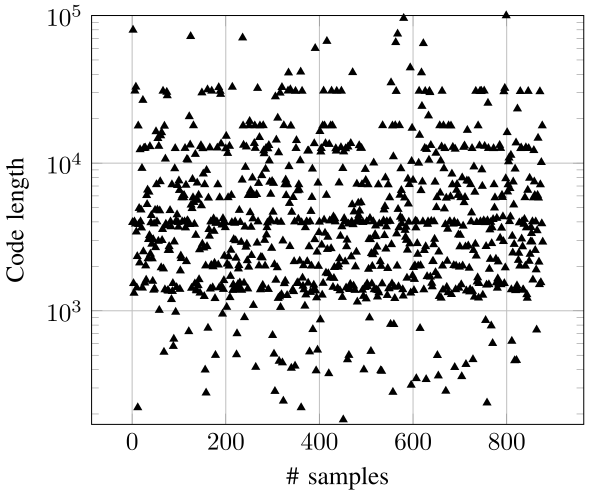
Before proceeding with the experiment, we collected Microsoft Office document files which contained VBA macros. Owing to the fact that our study targets VBA macros, we collected ".docm" and ".xlsm" files, which will likely contain macros, through keyword searches from Google. We also collected all the MS Office files that were classified as malicious in the malware portal [30]–[32] unconditionally, to ensure that our proposed method is well-suited to be applied to the malicious files. The sample collection was done from 2016 to 2017.

We verified the hash value of the collected files so that there were no duplicates, and we also excluded the files which did not have VBA macros. In the next step, we double-checked the detection results of the VirusTotal [32] and the VBA macros of files to determine the benign and malicious dataset, so that the only samples using VBA macros as an attack vector were included in the malicious dataset. As a result of the data collection, we obtained 2,537 files in which 773 are benign, and 1,764 are malicious. Table II displays the summary of our dataset with the average file size of each sample set. According to our observation, malicious files tend to be much smaller in terms of file size, which means that most of the attacks using VBA macros work to download malware from a remote address and execute it, and do not actually include malware in the file itself [15].

Although VirusTotal includes the results of about 60 different AV vendors who take advantage of individual detection mechanism, it is not 100% accurate. Because there is no conclusive criterion to determine a sample's maliciousness, we set a threshold to divide samples into malicious/benign training dataset. We set this threshold loosely to prevent the training samples from being mislabeled. In detail, we labeled a sample as malicious if more than 25 vendors detected it as malicious, and labeled it as benign if less than or equal to 2 vendors marked it as malicious. Every sample in between was manually inspected by three security researchers who specialize in VBA macros.



(a) Code length distribution of total 877 non-obfuscated VBA macros which are randomly selected from collected samples. The code length of non-obfuscated VBA macros is uniformly distributed that has no tendency between the samples.



(b) Code length distribution of total 877 obfuscated VBA macros. We can see the tendency that a group of VBA macros form a horizontal line which have similar code length of 1500, 3000, and 15000.

Fig. 5: Code length distribution of (a) non-obfuscated, and (b) obfuscated VBA macro samples. The x-axis indicates arbitrary sample in each dataset.

B. Preprocessing

The next step for detecting obfuscation is preprocessing. By preprocessing we mean to extract VBA macros from the collected MS Office document files, remove small (insignificant) and duplicated macros, and label training samples.

To obtain the VBA macros from Microsoft Office document file, we need to open the document file directly or parse the structure of OpenXML (OLE in the previous version of MS Office 2003). Given that malicious VBA macros are often executed when documents are opened, we use oletools in the extraction of VBA macro codes [33]. Oletools is an open source Python package to analyze Microsoft Office document files. It allows us to easily extract the VBA macros without opening the file.

Although we split our dataset into benign and malicious to provide the information about the relationship between maliciousness and obfuscation, the goal of this paper is to detect obfuscation in VBA macros. VBA macros in benign

TABLE II: Summary of collected MS Office document files.

Group	# by type		Avg. size	Collected from
	Word	Excel		
Benign dataset (773 in total)	75	698	1.1MB	Google
Malicious dataset (1,764 in total)	1,410	354	0.06MB	[30]–[32]
Total	1,485	1,052		

TABLE III: Summary of VBA macros extracted from MS Office files.

Group	# files	# macros	# obfuscated macros
Benign dataset	773	3,380	58 **(1.7%)**
Malicious dataset	1,764	832	819 **(98.4%)**
Total	2,537	4,212	877

datasets could be obfuscated, and vice versa. Therefore, we manually inspected and marked the macros with obfuscating features (described in Section III) as "obfuscated".

In this manual labeling process, we observed that the macros of less than 150 bytes are not meaningful, either malicious or benign, because they are only made up of comments or practice code that had no particular purpose. Therefore, insignificant macros with too short of a length were excluded from our dataset.

Table III shows that the majority of malicious VBA macros are obfuscated. Only 1.7% of the benign macros are obfuscated, whereas 98.4% of the malicious macros are obfuscated. With a huge gap of obfuscation rates in each of the dataset group, we verified the obfuscation tendency in benign and malicious macros: malicious macros are more likely to be obfuscated.

Also, there is a large gap in the number of extracted VBA macros. As explained in the data collection step, we already eliminated the duplicates ones, after collecting the Microsoft Office document files. But there is still a possibility that the files have macro duplicates. We found that there were about 5k macros for the overall dataset in this process of duplicates elimination. Finally, the number of macros was narrowed down to 3,380 and 832 respectively, in the benign and malicious dataset.

In the case of the benign dataset, the number of macros increases to more than 4 times as many as the number of files, because one file could have several macros. However, in the case of a malicious dataset, even though we only collected files that contain more than one macro in the data collection step, the number of macros is halved compared to the number of files. This means that most of the malicious documents which contains VBA macros are using the same macros.

In addition to this, we also examined the code length of the macros belonging to the non-obfuscated and obfuscated group. The results are shown in Figure 5 (a) and (b). Each figure displays the code length distribution in normal and obfuscated VBA macros, respectively. Figure 5 (a) is uniformly distributed throughout, this could also be evidence that our dataset is well-collected, including the informative benign macros. Alternatively, in Figure 5 (b), it can be seen that the macros are somewhat grouped to form several horizontal lines. Generally, we can expect that obfuscated code is reproduced with a custom obfuscater with different options. Especially in the malicious case, malware writers are expected to make variations to avoid the signature-based detection of AVs. We can interpret the results shown in Figure 5 (b), as the result of this expectation. This means that there are a large number of macros which have a similar code length even after the duplicate elimination.

TABLE IV: Summary of 15 static features used in our proposed method.

Features	Description	Used In:
V1	# of chars in code except comments	
V2	# of chars in comments	[24], [26]
V3	avg. length of words	[26]
V4	var. length of words	
V5	appearance frequency of string operators	[26]
V6	% of chars belonging to string	[26]
V7	avg. length of strings in code	[24], [26]
V8	% of text functions called	
V9	% of arithmetic functions called	
V10	% of type conversion functions called	
V11	% of financial functions called	
V12	% of functions with rich functionality called	
V13	Shannon entropy of the file	[26], [34]
V14	avg. length of identifiers	
V15	var. length of identifiers	

C. Feature selection

We summarized the types of obfuscation techniques in Section III. After observing the results of applying the obfuscation techniques, we built a set of features based on each of the obfuscation techniques. The proposed features are depicted in Table IV. Each of the features targets obfuscation, and some of them are from related studies. Given that four types of techniques have distinct characteristics, different combinations of features are required for an effective detection.

1) Detection of O1 (Random obfuscation): The O1 obfuscation technique randomizes the identifier in the macro code. The identifier refers to both the function names and variable names, and O1 can be applied to both of them. As a result of O1 obfuscation, the randomness of the macro code increases. To measure the randomness of macros, we use the Shannon entropy of the file as the feature V13 [35]. The computation of the entropy is performed on the basis of each character of the macro code. If p_i is considered to be the rate at which character *i* appears in the entire macro code, entropy *H* follows Shannon's Entropy formula.

$$H(X) = -\sum_{i} p_i \log_2 p_i$$

We use 2 additional features, V14 and V15 to capture the characteristics of O1. Because the identifiers with O1 techniques have various lengths, we calculate the length of the identifier. V14 is the average length of identifiers used in macro codes, V15 is the variance of each identifier length.

2) Detection of O2 (Split obfuscation): In the VBA macros with O2, more strings and string operators are observed than normal macros for the purpose of avoiding the detection of AVs. It also contains many unused dummy strings. For this type of obfuscation, we use V5-V7. V5 contains the number of occurrences of string operators such as '+', '=' or '&', which are used for string concatenation. Feature V6 is % of characters belonging to strings, and V7 calculates the average length of strings. These three features can indicate the unusual appearance of strings in obfuscated macros.

3) Detection of O3 (Encoding obfuscation): Encoding obfuscation is related to the use of various function calls. It is often used with O2, hiding keywords that can be detected by AVs, e.g., URL or .exe. It also uses infrequent financial functions which are only used for accounting and financial calculations to create more varied variants. To capture the characteristics of O3, we use V8-V11, while attempting to cover as many types as possible. The examples of the functions included for each feature are listed as follows. The rest of functions can be found by referring to the VBA language specification [27].

- **V8 (text functions)**: Asc(), Chr(), Mid(), Join(), InStr(), Replace(), Right(), StrConv(), etc.
- **V9 (arithmetic functions)**: Abs(), Atn(), Cos(), Exp(), Log(), Randomize(), Round(), Tan(), Sqr(), etc.
- **V10 (type conversion functions)**: CBool(), CByte(), CChar(), CStr(), CDec(), CUInt(), CShort(), etc.
- **V11 (financial functions)**: DDB(), FV(), IPmt(), PV(), Pmt(), Rate(), SLN(), SYD(), etc.

4) Detection of O4 (Logic obfuscation): O4 changes the entire shape of the targeted code by inserting dummy codes and reordering the code. As we mentioned in Section III, code reordering does not affect our proposed method as we use static features. We use V1-V4 to capture the dummy code insertion, which leads to an increase in code size. Before describing each feature, we use "words" to represent the units delimited by whitespace and VBA programming language symbols. "words" is used as a part of the features to detect maliciousness in [24]; it is also included in our features as V3 and V4 because it is a discriminant feature for dividing obfuscated and non-obfuscated code. V3 and V4 represent the average and the variance of "word" length, respectively.

To balance the effect of each feature on the training classifiers, a normalization process is required. Aebersold et al. [26] divided the value of features, which need to be normalized, by the length of the entire scripts. Instead, we assign the length of the comments-excluded macro code to V1, and the length of comments to V2. Then we use V1 as the normalization unit for more effective training.

V1-V11 and V13-V15 are selected to capture the characteristics of each obfuscation technique. In addition, there are a few unique functions observed in the obfuscated macros. Obfuscation is usually applied to code that has something to hide rather than tiny, insignificant code. Obfuscation is used to protect the intellectual property of the program code, or to hide malicious behavior in malware. In both cases, obfuscated code has a significant role that programmer wants to hide, hence it often leads to the use of certain functions with relatively rich functionality. For examples, the *Shell()* function is able to run executable programs, *CallByName()* can execute methods of objects which have full functionality in the VBA macro. Including these functions, V12 counts the use of functions that can write, download, or execute files.

D. Machine learning classifiers

We choose five different supervised machine learning classifiers to evaluate the performance of our proposed method: Random forest (RF), Support Vector Machine (SVM), Linear Discriminant Analysis (LDA), Bernoulli Naive Bayes (BNB), and Multi-Layer Perceptron (MLP). In addition to the four classifiers already used in previous studies [24], [26], we introduced the MLP classifier which is a class of artificial neural network models. We choose Scikit-learn [36] to use the aforementioned classifiers. Instead of describing the details of each classifier, we provide a customization parameter as well as a brief description of each classifier in this part of the paper.

Support Vector Machine (SVM) [37] finds the optimal, or maximum-margin hyperplane in a feature space that can separate a feature space into two classes (in our work, two classes indicate obfuscated and non-obfuscated). In our experiment, we use C=150, γ =0.03 as a parameter.

Random Forest (RF) [38] is an ensemble learning method for classification or regression. It constructs multiple decision trees in the training phase. It is known that Random Forest is less likely to have an overfitting problem than a decision tree [39].

Multi-Layer Perceptron (MLP) [40] is a feed-forward artificial neural network model that conducts supervised learning by backpropagation using one or more hidden layers between the input and output layer.

Linear Discriminant Analysis (LDA) [41], which is a form of supervised dimensionality reduction, is a generalization of Fisher's linear discriminant [42] that finds the linear subspace which maximizes the separation between two classes.

Naive Bayes [43] classifiers are a set of simple probabilistic classifiers based on applying the Bayes' Theorem with naive independence assumptions between the features used. We use Bernoulli Naive Bayes (BNB) in the evaluation of proposed method.

TABLE V: Evaluation results of proposed approach.

Feature set	Classifier	Accuracy	Precision	Recall
V1-V15	SVM	0.955	0.881	0.906
	RF	0.965	**0.982**	0.848
	MLP	**0.970**	0.938	**0.915**
	LDA	0.901	0.842	0.64
	BNB	0.891	0.75	0.713
J1-J20	SVM	0.753	0.445	0.751
	RF	**0.903**	**0.841**	0.657
	MLP	0.834	0.76	0.316
	LDA	0.826	0.677	0.318
	BNB	0.701	0.391	**0.775**

V. Evaluation

In this section, the evaluation results based on the method proposed in section IV will be described. We extracted the feature matrix from the preprocessed dataset with the features introduced in Table IV. After the five different classifiers have undergone the training process, we will evaluate the classification performance with several evaluation metrics. Before going into the details of evaluation, we briefly explain the evaluation metrics to be used in this section.

For more precise and quantitative measures of our classification performance, we use several evaluation metrics: Accuracy, Precision, Recall, F_β score, and AUC of ROC curve. We use accuracy, precision and recall to evaluate the basic classification performance, and choose β=2 of the F_β score to emphasize the security aspect. F_2 score is often used when weighing recall more than precision. By putting an emphasis on recall, we can make sure malicious VBA macro is not executed on the users' system. In addition, we use the Receiver Operating Characteristic (ROC) curves and Area Under the Curve (AUC), which is the one of the standard convention, to show the comparison of classification results in a more intuitive manner.

We used 4,212 macros for the evaluation of classification performance, 877 of which are marked as obfuscated. Although our dataset is large enough to evaluate the classification performance of the proposed method, we use 10-fold Cross Validation (CV) to improve the statistical reliability. Therefore, the experimental results to be described below are the results of applying the 10-fold cross validation.

Table V shows the classification results with basic evaluation metrics. The feature set we proposed is marked as V1-V15 in the leftmost column. As a result of the evaluation, SVM, RF and MLP classifiers show relatively high performance among five classifiers. In particular, RF recorded a precision of 98.2% and MLP recorded a recall of 91.5%. However, LDA and BNB classifiers were found to be inadequate for detecting obfuscated VBA macro.

The evaluation result with F_2 score is depicted in Figure 6. The result of the proposed method is the bars labeled 'V feature set'. Because obfuscation detection is primarily concerned with security purposes, we emphasize recall to minimize false negatives. As MLP classifier showed relatively high performances in the basic three metrics, accuracy, precision, and recall, it also recorded the highest F_2 score of 92%. In a related study that evaluated detection performance with the F_2 score [24], we can see that our method is 11.4% higher, given that 80.6% was its maximum.

We can then ask ourselves the following research question: "It has been confirmed that the proposed features and classification method are effective in detecting obfuscated VBA macro, but how effective would it be to use the malware detection features of the related studies that have already been

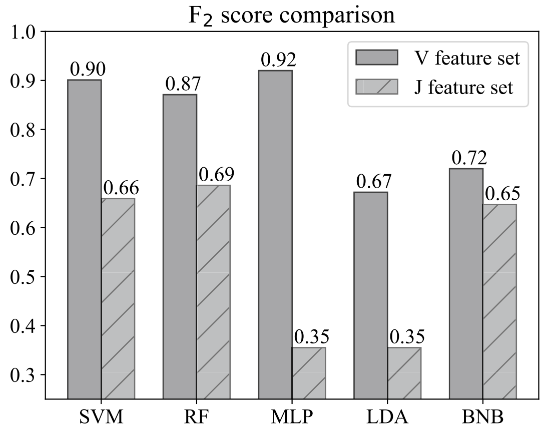


Fig. 6: The results of machine learning classification using the proposed feature set are expressed as F_2 score. When using the MLP classifier, the result was the highest at 92%.

TABLE VI: Summary of the features used in related work.

Features	**Description**	**Used In:**
J1	length in characters	[24], [26]
J2	avg. # of chars per line	[24], [26]
J3	total number of lines	[24], [26]
J4	# of strings	[24]
J5	% human readable	[24]
J6	% whitespace	[24], [26]
J7	% of methods called	[24]
J8	avg. string length	[24], [26]
J9	avg. argument length	[24], [26]
J10	# of comments	[24], [26]
J11	avg. comments per line	[24]
J12	# words	[24]
J13	% words not in comments	[24]
J14	% of lines $>$ 150 chars	[26]
J15	Shannon entropy of the file	[26], [34]
J16	share of chars belonging to a string	[26]
J17	% of backslash characters	[26]
J18	avg. # of chars per function body	[26]
J19	% of chars belonging to a function body	[26]
J20	# of function definitions divided by J1	[26]

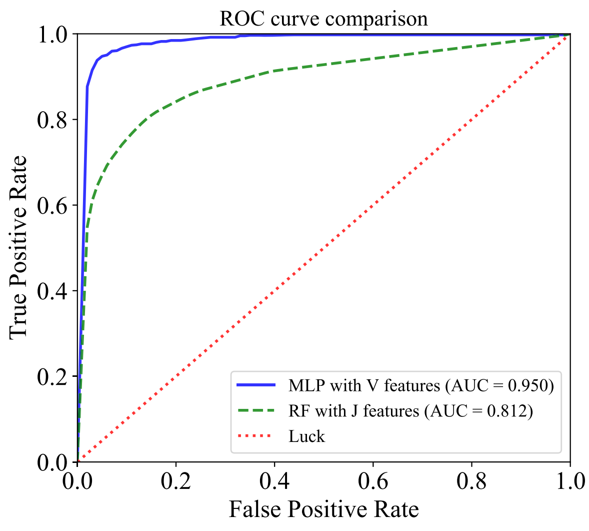


Fig. 7: The solid curve and dashed curve represents ROC curves of MLP classifier with proposed feature set and RF classifier with comparison feature set, respectively.

conducted? Would it not be more effective?". In response to this question, we added a comparative experiment to detect obfuscated VBA macros using the same machine learning approach to the same dataset. The features used in related studies [24], [26] are listed in Table VI.

Due to the linguistic differences between JavaScript and Visual Basic for Applications, many of the features used in obfuscated JavaScript detection are not applicable for obfuscated VBA macro detection. For example, "# of *eval()* calls divided by entire code length" was used in the related paper [26], which was not implemented in this study because it is difficult to match the *eval()* function to corresponding VBA function. Besides, J14, originally '% of lines with more than 1000 characters', was modified to reflect the characteristics of VBA macros that can not be applied the minification technique of removing linefeed. The results of this comparison experiment are shown in Table V and Figure 6 as 'J feature set'.

Table V includes the evaluation result of comparison experiment (marked as J1-J20). The accuracy and precision of RF classifier were the highest at 90.3% and 84.1% among five classifiers, respectively. However, in all aspects, the classification performance was much better when using V features, than when using J features. In order to comprehensively evaluate the classification performance, we introduced the F_2 score and the result is depicted in Figure 6. The maximum F_2 score was found in the MLP classifier for V feature set (0.92) and the RF classifier for J feature set (0.69).

As another comprehensive evaluation method, the AUC of ROC curves were calculated. Figure 7 shows the ROC curves of MLP and RF, which scored maximum F_2 for proposed V and J features, respectively. MLP classifier with proposed feature set (V features) has an AUC of 0.95, and comparison experiment (J features) gets 0.812. It shows that our proposed method outperformed the previous studies by 0.138 on the AUC basis.

As a result, we obtained up to 92.0% F_2 score with proposed feature set when obfuscation detection was performed using the MLP classifier. This is 23% higher than the result of using the features proposed in the related studies. The accuracy, precision, and recall show better results, and the AUC value of the ROC curve was 0.950, showing that the proposed method and features are suitable for obfuscated VBA macro detection.

VI. Discussion

A. *Obfuscation detection and malicious code detection*

We presented 15 static features for obfuscation detection, and evaluated our proposed method using various evaluation metrics. However, this is a method for obfuscation detection, not malicious code detection. We investigated a sufficient number of MS Office document files to clarify the relationship between obfuscation and maliciousness. This obfuscation detection method can play a major role in malicious code detection, as the rate of obfuscation applied differs greatly between malicious dataset (98.4%) and benign dataset (1.7%) as described in Table III.

Currently, the distinction between malicious code detection and obfuscated code detection is unclear in malware detection research. As long as cases where obfuscation techniques used to protect intellectual property rights exist, malicious code detection should be distinguished from obfuscated code detection. However, a few of the related studies used the characteristics of obfuscation to detect malicious codes without considering obfuscation techniques [18], [24]. The confusion between maliciousness and obfuscation may lead to an increase in false alarms. Therefore, we generally classified obfuscation type (O1-O4) to prevent this mistake, and designed the feature set to not be biased towards the characteristics of a specific obfuscation tool.

In order to address the need for a counteraction against the increasing obfuscated VBA macro malware, we compared the ability of J feature set and our proposed V feature set regarding obfuscation detection. The results showed that the J feature set underperformed against the proposed V feature set, but this does not mean that the research results regarding JavaScript is bad. Rather, in regards to detection of obfuscation in highly obfuscated VBA macro malware (98.4%), applying existing studies (J feature set)—that does not take into account the characteristics of obfuscation—is not ideal.

B. *Case studies: anti-analysis techniques in VBA*

The obfuscation techniques observed in VBA macros are categorized into four types (O1-O4) in Section III. When using features based on the O1-O4, we succeeded in identifying obfuscation with an accuracy of 97%. In addition to obfuscation, however, several tricks have been found for the purpose of hindering the analysis and understanding of the code. In this

```
Private Sub Document_Open()
  UYjwCZdgnz = ActiveDocument.Variables("waGnXV").Value()
  mambaFRUTISsIn = UserForm1.Label1.Caption
  Shell UYjwCZdgnz, 0
  Shell mambaFRUTISsIn, 0
End Sub
```

(a) A sample macro code which uses hiding string data. If the code analyst has only the above code, it can not be determined whether it is malicious or not before checking what 'UYjwCZdgnz' and 'mambaFRUTISsIn' contain.

```
 1 Public Sub RemoveIDAndFormatRow()
 2   shtiletMurinoASALLLP = acs.responseBody
 3   ProjectAndNow.Write shtiletMurinoASALLLP
 4   CoachesReport ""
 5   Exit Sub
 6   Rows.Select
 7   'Broken code here
 8   Sel.ection.RowHeight = 15
 9   Colu.mns("A:A").Delete
10   Colu.mns("A").ColumnWidth = 25
11   Colu.mns("C").ColumnWidth = 24.71
12   Colu.mns("I:R").ColumnWidth = 11
13 End Sub
```

(b) Inserting broken code causes an error when code parser tries to interpret "Sel" or "Colu" nonexistent objects.

Fig. 8: Example code of anti-analysis technique

paper, we call these tricks to hinder code analysis as anti-analysis technique and distinguish it from obfuscation technique. Obfuscation (O1-O4) is used generically in scripting code and makes a significant difference in the appearance of existing code. However, anti-analysis technique is limited in scope that can be applied to code and is designed to prevent specific analysis method.

The anti-analysis techniques to be introduced are not directly addressed or included in the proposed method. However, they also interfere with the process of analyzing the code and tend to be found together in obfuscated VBA macros. For further malware detection research, we organize the basic anti-analysis techniques observed in VBA macro as follows: 1) Hiding string data, 2) Inserting broken code, and 3) Changing the flow.

1) Hiding string data: Microsoft Office documents provide useful data spaces for storing string data. For example, one can store string data as the document's property value, the Caption value of CommandButton, Label, and Form controls, or the ControlTipText value of UserForm controls [44]. If a malware writer hides malicious string values in these fields or even in the cell value of an Excel document and the malware refers to them, this prevent the use of static analysis techniques which analyze the VBA macro source code. Figure 8 (a) shows the case of hiding string data technique.

2) Inserting broken code: This technique is frequently adopted in obfuscated VBA macros. It is done by inserting broken code which causes run-time error. However, as Figure 8 (b) shows, the instruction pointer actually exits in line number 5, before reaching the broken code starting from line number 8. So this anti-analysis technique does not affect the actual behavior of the macro code, but it is considered as a syntax error when trying to parse the code.

3) Changing the flow: Another anti-analysis strategy, which can be used together with the aforementioned anti-analysis techniques, is achieved by switching the execution flow. It is done by using a conditional branching statement, together with checking certain condition is satisfied. Certain condition may be an http response code that verifies that the connection is well established, or it may be the number of recently opened files to prevent sandboxing analysis [45].

VII. Conclusion

This paper is the first research to propose obfuscated VBA macro detection using machine learning method. Attacks using VBA macro have been increasing since 2014. Given the familiarity of the MS Office document, this type of attack should not be taken lightly. Even though AV agencies are increasingly reporting attacks using VBA macro, little research has been conducted to mitigate them.

Unlike the conventional malware which exploits the vulnerability of programs, attacks using VBA macro utilize legitimate functions provided by MS Office document. These threats are not caused by a programmers mistake, nor are mitigated by a security update. A general way to avoid this kind of cyber attack is to improve the security awareness of the end users. It includes: not downloading attachments from untrusted e-mails, and recognizing the potential damage that even one malicious document can bring.

Research on identifying obfuscation techniques, which are applied to VBA macros in the document, is one of the countermeasures to prevent malware infection before malicious code is executed. We collected 4,212 benign and malicious VBA macros to investigate how many macros were obfuscated. 98.4% of the malicious macros files were obfuscated, one the other hand, only 1.7% of the benign macros were obfuscated.

In this paper, we proposed obfuscated VBA macro detection with machine learning based approach. We have classified VBA macro obfuscation techniques into four types and introduced a feature set for effective obfuscation detection. In the process of selecting detection features, several features were selected from JavaScript related studies after being modified to reflect the characteristics of VBA macro, or excluded if not applicable for VBA macro. We then evaluated the classification result of the five suggested machine learning classifiers using various evaluation metrics. The evaluation results demonstrated that our detection approach achieved a F_2 score improvement of greater than 23% compared to those of related studies.

Acknowledgment

The authors would like to express our sincere gratitude for our shepherd, Eric Eide, and the anonymous reviewers for their every valued comments to improve the quality of the paper. This research has been supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (NO.2017-0-00184, Self-Learning Cyber Immune Technology Development).

References

[1] F-Secure, "virus:w32/concept description," https://www.f-secure.com/v-descs/concept.shtml, 2017.

[2] M. Hypponen *et al.*, "Threat report 2015," F-Secure Corporation, https://www.f-secure.com/documents/996508/1030743/Threat_Report_2015.pdf, Tech. Rep., 2015.

[3] D. Chi, "Microsoft office 2000 and security against macro viruses," Symantec Antivirus Research Center, https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/microsoft-2000-security-against-macro-99-en.pdf, Tech. Rep., 2000.

[4] G. Szappanos, "VBA is not dead!" https://www.virusbulletin.com/virusbulletin/2014/07/vba-not-dead, June 2014.

[5] G. Chantry, "From the labs: VBA is definitely not dead in fact, its undergoing a resurgence," https://nakedsecurity.sophos.com/2014/09/17/vba-injectors/, September 2014.

[6] P. Wood *et al.*, "Symantec internet security threat report vol 21," Symantec Corporation, Tech. Rep., 2016.

[7] D. Gudkova, M. Vergelis, and N. Demidova, "Spam and phishing in q3 2016," *AO Kapersky Lab*, 2016.

[8] C. Beek *et al.*, "Mcafee labs threats report," McAfee Inc., Santa Clara, CA., https://www.mcafee.com/ca/resources/reports/rp-quarterly-threats-sept-2017.pdf, Tech. Rep., september 2017.

[9] W. Xu, F. Zhang, and S. Zhu, "The power of obfuscation techniques in malicious javascript code: A measurement study," in *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on*. IEEE, 2012, pp. 9–16.

[10] Z. Tzermias, G. Sykiotakis, M. Polychronakis, and E. P. Markatos, "Combining static and dynamic analysis for the detection of malicious documents," in *Proceedings of the Fourth European Workshop on System Security*. ACM, 2011, p. 4.

[11] S. Jana and V. Shmatikov, "Abusing file processing in malware detectors for fun and profit," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 80–94.

[12] T. Schreck, S. Berger, and J. Göbel, "Bissam: Automatic vulnerability identification of office documents," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2012, pp. 204–213.

[13] C. Smutz and A. Stavrou, "Preventing exploits in microsoft office documents through content randomization," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2015, pp. 225–246.

[14] K. Iwamoto and K. Wasaki, "A method for shellcode extraction from malicious document files using entropy and emulation," *International Journal of Engineering and Technology*, vol. 8, no. 2, p. 101, 2016.

[15] M. Mimura, Y. Otsubo, and H. Tanaka, "Evaluation of a brute forcing tool that extracts the rat from a malicious document file," in *Information Security (AsiaJCIS), 2016 11th Asia Joint Conference on*. IEEE, 2016, pp. 147–154.

[16] A. Cohen, N. Nissim, L. Rokach, and Y. Elovici, "Sfem: Structural feature extraction methodology for the detection of malicious office documents using machine learning methods," *Expert Systems with Applications*, vol. 63, pp. 324–343, 2016.

[17] N. Nissim, A. Cohen, and Y. Elovici, "Aldocx: detection of unknown malicious microsoft office documents using designated active learning methods based on new structural feature extraction methodology," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 3, pp. 631–646, 2017.

[18] E. Gaustad, "Applied Machine Learning: Defeating Modern Malicious Documents," https://www.rsaconference.com/events/us17/agenda/sessions/6662-applied-machine-learning-defeating-modern-malicious, February 2017.

[19] Y. Choi, T. Kim, S. Choi, and C. Lee, "Automatic detection for javascript obfuscation attacks in web pages through string pattern analysis," in *Int'l Conference on Future Generation Information Technology*, 2009, pp. 160–172.

[20] W. Xu, F. Zhang, and S. Zhu, "Jstill: mostly static detection of obfuscated malicious javascript code," in *ACM conference on Data and application security and privacy*, 2013, pp. 117–128.

[21] D. Liu, H. Wang, and A. Stavrou, "Detecting malicious javascript in pdf through document instrumentation," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*. IEEE, 2014, pp. 100–111.

[22] K. Kim, I. L. Kim, C. H. Kim, Y. Kwon, Y. Zheng, X. Zhang, and D. Xu, "J-force: Forced execution on javascript," in *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2017, pp. 897–906.

[23] M. AbdelKhalek and A. Shosha, "Jsdes: An automated de-obfuscation system for malicious javascript," in *Proceedings of the 12th International Conference on Availability, Reliability and Security*. ACM, 2017, p. 80.

[24] P. Likarish, E. Jung, and I. Jo, "Obfuscated malicious javascript detection using classification techniques." in *MALWARE*, 2009, pp. 47–54.

[25] M. Jodavi, M. Abadi, and E. Parhizkar, "Jsobfusdetector: A binary pso-based one-class classifier ensemble to detect obfuscated javascript code," in *Artificial Intelligence and Signal Processing (AISP), Int'l Symp. on*, 2015, pp. 322–327.

[26] S. Aebersold, K. Kryszczuk, S. Paganoni, B. Tellenbach, and T. Trowbridge, "Detecting obfuscated javascripts using machine learning," in *Proceedings of the 11th International Conference on Internet Monitoring and Protection (ICIMP)*, 2016.

[27] *[MS-VBAL]: VBA Language Specification*, Microsoft Corporation, https://msdn.microsoft.com/en-us/library/dd361851.aspx, December 2016.

[28] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1997.

[29] "Crunchcode: the obfuscator for vba macros," http://www.crunchcode.de/.

[30] Malwr, "Malwr - malware analysis by cuckoo sandbox," https://malwr.com, 2016.

[31] VirusShare, "Virusshare.com - because sharing is caring," https://virusshare.com, 2016.

[32] VirusTotal, "Virustotal - free online virus, malware and url scanner," https://www.virustotal.com, 2016.

[33] oletools, "oletools - python tools to analyze ole and ms office files," https://www.decalage.info/python/oletools.

[34] B.-I. Kim, C.-T. Im, and H.-C. Jung, "Suspicious malicious web site detection with strength analysis of a javascript obfuscation," *International Journal of Advanced Science and Technology*, vol. 26, pp. 19–32, 2011.

[35] C. E. Shannon, "A mathematical theory of communication," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 5, no. 1, pp. 3–55, 2001.

[36] Scikit-learn, "scikit-learn - machine learning in python," http://scikit-learn.org/.

[37] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[38] H. T. Kam, "Random decision forest," in *Proc. of the 3rd Int'l Conf. on Document Analysis and Recognition, Montreal, Canada, August*, 1995, pp. 14–18.

[39] T. Hastie, R. Tibshirani, and J. Friedman, "The elements of statistical learning 2nd edition," 2009.

[40] S. S. Haykin, S. S. Haykin, S. S. Haykin, and S. S. Haykin, *Neural networks and learning machines*. Pearson Upper Saddle River, NJ, USA:, 2009, vol. 3.

[41] S. Mika, G. Ratsch, J. Weston, B. Scholkopf, and K.-R. Mullers, "Fisher discriminant analysis with kernels," in *Neural Networks for Signal Processing IX, 1999. Proceedings of the 1999 IEEE Signal Processing Society Workshop*. IEEE, 1999, pp. 41–48.

[42] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of human genetics*, vol. 7, no. 2, pp. 179–188, 1936.

[43] A. McCallum, K. Nigam *et al.*, "A comparison of event models for naive bayes text classification," in *AAAI-98 workshop on learning for text categorization*, vol. 752. Madison, WI, 1998, pp. 41–48.

[44] *[MS-OFORMS]: Office Forms Binary File Formats*, Microsoft Corporation, https://msdn.microsoft.com/en-us/library/cc313125(v=office.12).aspx.

[45] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using cwsandbox," *IEEE Security & Privacy*, vol. 5, no. 2, 2007.