Investigating Representations of Obfuscated Malicious PowerShell

by Carolyn J. Holz

S.B., C.S. Massachusetts Institute of Technology (2015)

Submitted to the

Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

June 2019

Author: _____
Department of Electrical Engineering and Computer Science
May 24, 2019

Certified by: _____
Una-May O'Reilly, Principle Research Scientist, Thesis Supervisor
May 24, 2019

Certified by: _____
Erik Hemberg, Research Scientist, Thesis Co-Supervisor
May 24, 2019

Accepted by: _____
Katrina LaCurts, Chair, Master of Engineering Thesis Committee

Investigating Representations of Obfuscated Malicious PowerShell

by Carolyn J. Holz
Submitted to the Department of Electrical Engineering and Computer Science
May 24, 2019

In partial fulfillment of the requirements for the degree of Master of Engineering in

Electrical Engineering and Computer Science

Abstract

PowerShell is a popular scripting language due to its widespread use and access to critical system functions. However, these factors also contribute to its popularity amongst malware creators. In addition to the extensive access they can achieve with PowerShell, attackers can also obfuscate their PowerShell to make it more difficult to detect. Current detection methods rely on detecting signatures of known malicious scripts which can be easily broken with simple obfuscations. This work seeks to find a more abstract representation of script functionality using Abstract Syntax Trees so that an unseen obfuscated script can be detected if a related script is already known malware. We determine that simple AST based features such as node count and depth along with distance measures calculated from the node types and node orders within the AST are fairly sufficient to attribute obfuscated scripts to their originating script.

Thesis Supervisor: Una-May O'Reilly
Title: Principle Research Scientist

Thesis Supervisor: Erik Hemberg
Title: Research Scientist

Acknowledgments

Many thanks to Erik Hemberg for his guidance and patience throughout this year. Thank you for providing suggestions about paper structure, next steps for analysis, and sticky bike locks.

Thanks to Abdullah Al-Dujaili for providing direction in finding related work and literature as well as answering all of my questions.

Thanks to ALFA Group and all its members for allowing me the privilege of working with them this year. Thank you all for providing a comfortable and friendly space to work and share ideas and for making my MEng experience especially enjoyable.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Security and malware detection are a major concern facing computer scientists and users. Attackers create malicious software and scripts which exploit vulnerabilities in the operating system or other programs in order to gain access to sensitive data and critical system functions. They may steal or ransom the data or cause the system to malfunction, impairing the computer's usability. This is especially concerning in cases where the targeted software is extremely common or when the attack occurs on a large network of machines which allow it to spread quickly. Defenders look for ways to detect and stop these attacks before they can start. This can be done by static analysis of the program via signatures of known attacks or by using machine learning models.

PowerShell has become a popular scripting language due to its flexibility and access to operating system services such as the file system and registry keys [10]. It is pre-installed on Windows and has gained popularity on other operating systems over the past few years with its cross-platform version. However, its power and widespread use makes it a great tool and target for malicious attacks. Attackers are able to execute their malicious programs in PowerShell without installing them on the machine, leaving little evidence of their activities [4]. Commands can also be converted into non human-readable encodings, such as binary, in order to prevent detection of well-known malicious scripts upon visual inspection. Traditional detection methods

calculate a signature for known malware that is used to detect that attack in the future. The signature is created from static analysis of the original file and often cannot be used to detect similar scripts with the same function. The attacker can simply use obfuscation to break the signature. Obfuscation is the manipulation of a script or piece of code that changes the code signature without changing its function. This means that a defender must not only identify an existing attack and then create a signature to detect it but they must do this for all slightly modified or obfuscated attacks with the same essential function. In addition, freely available obfuscation tools make it easy to modify a script enough to get past traditional malicious PowerShell detection [1].

The fact that obfuscation is common in malware attacks might suggest that simply detecting obfuscation is a suitable method of malware detection. However, PowerShell may be obfuscated for benign as well as malicious reasons. A company or programmer may want to protect intellectual property by obscuring the original program so that another programmer cannot easily understand and duplicate it. This means that if we simply detect obfuscation we may reject benign scripts as well as useful ones. On the other hand, the purpose of obfuscating malicious PowerShell is to create enough difference between the obfuscated file and the original that the obfuscated file cannot be detected as malicious. Often attackers will apply multiple types of obfuscation to a script in order to increase this distance. An obfuscated script which breaks malware signature detection is called an adversarial example, which finds a blindspot in a model for malicious Powershell detection and exploits it to bypass the detector. We can attempt to find this blindspot by determining a distance measure that is maximized where a malicious PowerShell is able to evade detection [6]. If we can find this blindspot we can also correct for it by augmenting a malicious dataset with automatically generated adversarial examples and training new models which are robust to them. For instance, we could learn a model which classifies an adversarial example as originating from a particular malware script and prevent it from being run.

Due to these factors, there is emerging research into applying machine learning to the problem of malicious PowerShell detection. The benefit of using machine learning is that models can be trained to recognize the structure and function of a program which means we do not need to manually create a signature for new malware or even identify new attacks ahead of time in order to detect them. Current machine learning exploration for detecting malicious PowerShell includes NLP and convolutional neural network methods as well as Abstract Syntax Tree (AST) based structural analysis of programs[2]. We explore this last method and find a representation of PowerShell that can accurately describe the function of the program.

It has been shown that simple features of a PowerShell AST can be used to accurately classify malware families [8]. AST are data structure representations of programs which describe the structure of the program, such as the one shown in 1-1. AST nodes represent functional blocks in the program such as variable assigment while AST branches represent the control flow of the program such as if statements and for loops. Previous work identifies node types, subtree node count, and depth as features that can help classify families of malware. This suggests that including more information about AST structure as features could reveal more about script functionality. In addition, the features mentioned in previous work could be applied to other problems such as associating obfuscations with original scripts, since the classification tasks are similar.

1.1 Research Questions

The core aim of this research is to identify a useful representation for the function of PowerShell, independent of various obfuscation methods and manipulations of the code.

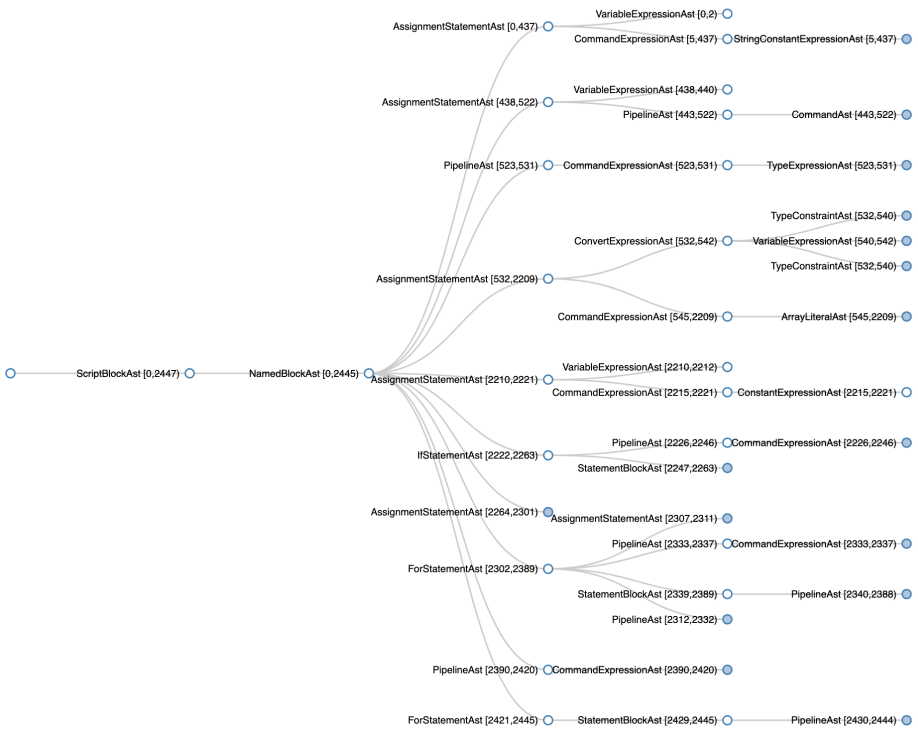


Figure 1-1: Visual example of PowerShell AST: Each node type is named along with the lines it encompasses. Child nodes are shown connected to their parents with lines between the nodes.

1.1.1 Exploring Representations

First, we would like to know what PowerShell scripts look like, with and without obfuscation. Specifically, what can we deduce from looking at a script visually? What can static inspection learn about a script? We expect that static inspection can learn a lot about the function of a script because it examines the structure of the script as well as its characters. We explore this visually by reading and comparing PowerShell in our dataset with obfuscations we create with freely available tools. We explore this quantitatively by creating character and AST representations of our dataset and obfuscations and comparing them with similarity and difference measures.

1.1.2 Measuring Difference

Next, we would like to compare our obfuscated and non-obfuscated PowerShell in a useful way. How can we represent scripts such that an obfuscated script can be attributed to its non-obfuscated counterpart? We expect to find solutions in the AST representation of PowerShell since this is a simple way of describing the structure of a script. We explore distance measures calculated from AST features to find measures which are robust to obfuscation.

1.1.3 Detecting Difference

Finally, we ask ourselves if our representations of PowerShell can accurately measure difference in a way that is useful for detection. We determine a set of features based on our distance measures which best represent the function of the program. We explore the usefulness of these measures using simple models to assign obfuscations to their original PowerShell.

1.2 Contribution

In this work, we will describe and implement a method for creating an augmented dataset of obfuscated PowerShell including original file labelling. We will also provide

extensive analysis of the dataset and several features for representing the structure of PowerShell. We will demonstrate the usefulness of these features by training models to assign obfuscated PowerShell to its original script and implement methods for extracting these features.

1.3 Document Structure

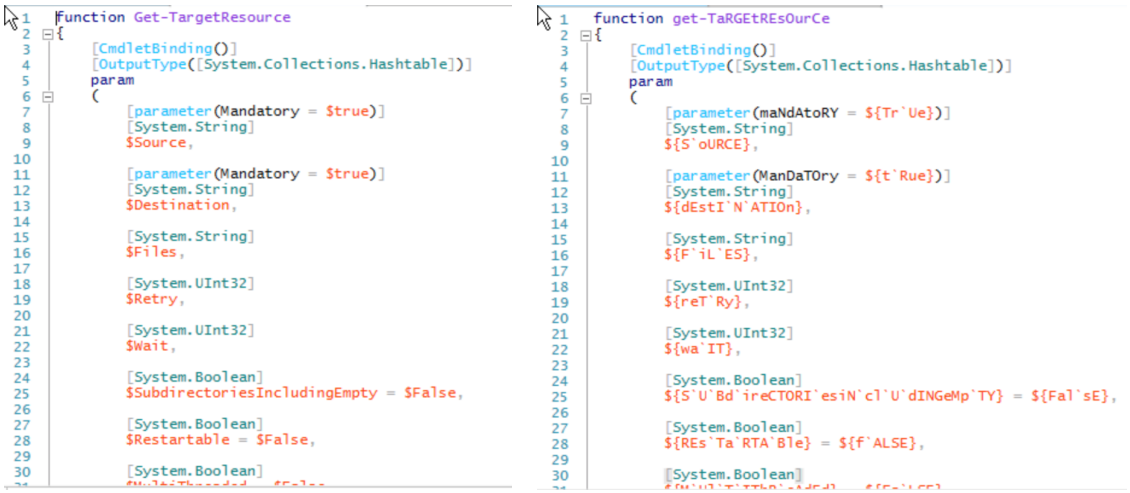
In the remaining sections we will discuss the following topics: first, existing tools for obfuscating PowerShell as well as the most recent methods of detecting malicious PowerShell in Chapter 2. Next, we describe the dataset in Chapter 3 and the methods we use to explore the effect of different obfuscation methods on PowerShell scripts in Chapter 4. We discuss the most useful features obtained from our exploration and their ability to model script functionality in Chapter 5. Finally, we discuss the accuracy of our models and further exploration needed to improve them in Chapter 6.

Chapter 2

Related Work

2.1 Obfuscation Tools

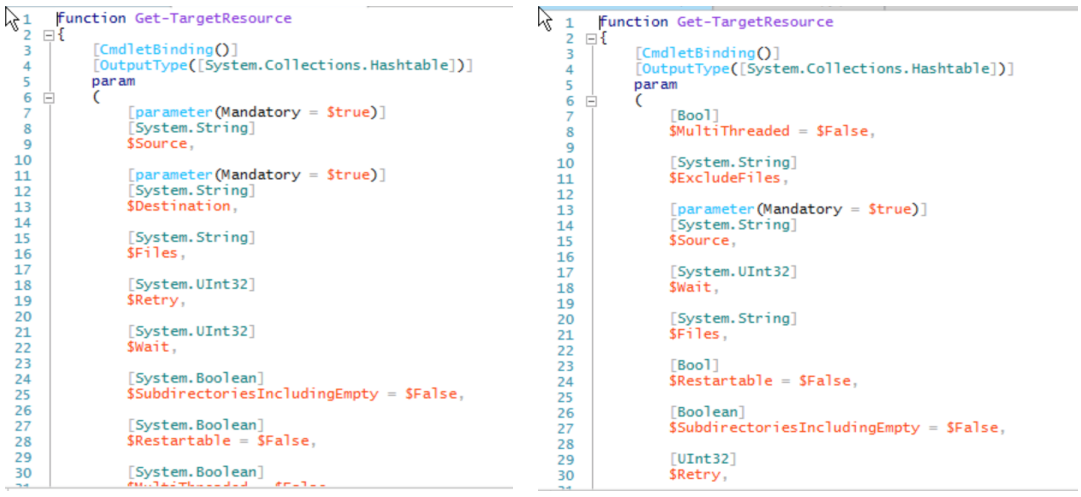
PowerShell obfuscation can be described in a few categories, token based, AST based, encoding, compression, launcher and cradle obfuscation. We focus on freely available obfuscation tools including `Invoke-Obfuscation`, `Invoke-CradleCrafter` and `ISESteroids` each of which perform a few of these types of obfuscation. Token or character based obfuscation changes the characters in a script and is often applied only to one token type at a time such as variables or command names. Token based obfuscation is also the most visually apparent, as shown in Figure 2-1. AST based obfuscation manipulates the structure of the script by reordering interchangeable elements such as variable assignments, as shown in Figure 2-2, or parallel method calls. Scripts can also be obfuscated by encoding them, for example as hex or binary strings, or by compressing them into a one line command. Finally, launcher and cradle techniques change the way a command is run. Launcher obfuscation creates a separate command or executable to be launched from another tool such as Python or the native command line and cradle obfuscation creates a new command which downloads the original command from another source. Both of these obfuscation types create potential for the command to be invoked unbeknownst to the user and run without leaving a trace on the local machine.



```powershell
function Get-TargetResource
{
    [CmdletBinding()]
    [OutputType([System.Collections.Hashtable])]
    param
    (
        [parameter(Mandatory = $true)]
        [System.String]
        $Source,

        [parameter(Mandatory = $true)]
        [System.String]
        $Destination,

        [System.String]
        $Files,

        [System.UInt32]
        $Retry,

        [System.UInt32]
        $Wait,

        [System.Boolean]
        $SubdirectoriesIncludingEmpty = $False,

        [System.Boolean]
        $Restartable = $False,

        [System.Boolean]
```

```powershell
function get-TaRGEtREsOurCe
{
    [CmdletBinding()]
    [OutputType([System.Collections.Hashtable])]
    param
    (
        [parameter(maNdAtoRY = ${Tr`Ue})]
        [System.String]
        ${S`oURCE},

        [parameter(ManDaTOry = ${t`Rue})]
        [System.String]
        ${dEstI`N`ATIOn},

        [System.String]
        ${F`iL`ES},

        [System.UInt32]
        ${reT`Ry},

        [System.UInt32]
        ${wa`IT},

        [System.Boolean]
        ${S`U`Bd`ireCTORI`esiN`cl`U`dINGeMp`TY} = ${Fal`sE},

        [System.Boolean]
        ${REs`Ta`RTA`Ble} = ${f`ALSE},

        [System.Boolean]
```

Figure 2-1: Visual Example of Token Obfuscation of Arguments: Character casing is changed and apostrophes have been inserted in the strings.



```powershell
function Get-TargetResource
{
    [CmdletBinding()]
    [OutputType([System.Collections.Hashtable])]
    param
    (
        [parameter(Mandatory = $true)]
        [System.String]
        $Source,

        [parameter(Mandatory = $true)]
        [System.String]
        $Destination,

        [System.String]
        $Files,

        [System.UInt32]
        $Retry,

        [System.UInt32]
        $Wait,

        [System.Boolean]
        $SubdirectoriesIncludingEmpty = $False,

        [System.Boolean]
        $Restartable = $False,

        [System.Boolean]
```

```powershell
function Get-TargetResource
{
    [CmdletBinding()]
    [OutputType([System.Collections.Hashtable])]
    param
    (
        [Bool]
        $MultiThreaded = $False,

        [System.String]
        $ExcludeFiles,

        [parameter(Mandatory = $true)]
        [System.String]
        $Source,

        [System.UInt32]
        $Wait,

        [System.String]
        $Files,

        [Bool]
        $Restartable = $False,

        [Boolean]
        $SubdirectoriesIncludingEmpty = $False,

        [UInt32]
        $Retry,
```

Figure 2-2: Visual Example of AST Obfuscation of Assignment Blocks: The order of parameter assignment is changed.

`Invoke-Obfuscation` [3] is a powerful open source obfuscation tool. It includes commands for many categories of obfuscation including:

- `TOKEN`: uses string manipulations such as concatenation, random casing, re-ordering with formatting methods, and adding tick marks in the string for the following token types:
  - `STRING, COMMAND, ARGUMENT, MEMBER, VARIABLE, TYPE, COMMENT, WHITESPACE`
- `AST`: reorders the following AST node types:
  - `NamedAttributeArgumentAst, ParamBlockAst, ScriptBlockAst, AttributeAst, BinaryExpressionAst, HashtableAst, CommandAst, AssignmentStatementAst, TypeExpressionAst, TypeConstraintAst`
- `ENCODING`: encodes the entire command as either:
  - `ASCII, HEX, OCTAL, BINARY, SECURE STRING (AES), BXOR, SPECIAL CHARACTERS, WHITESPACE`
- `COMPRESS`: convert the command to a one line command and compress to base 64
- `LAUNCHER`: convert command to an executable file of the following types:
  - `PS, CMD, WMIC, RUNDLL, VAR+, STDIN+, CLIP+, VAR++, STDIN++, CLIP++, RUNDLL++, MSHTA++`

`Invoke-CradleCrafter` [4] is an extension of `Invoke-Obfuscation` which focuses on obfuscating PowerShell as remotely downloaded executables. The scripts will either be downloaded to memory or disk in several languages and forms:

- `MEMORY`: Downloads into memory as `PS, .NET, CERTUTIL` in formats:
  - `STRING` - the command as a single string

  - `WEBREQUEST` - a readable stream, byte array, or structured data
  - `COMOBJECTS` - objects which are downloaded by Windows program interactions
  - `CSHARP` - compiled C# either inline or beforehand
- `DISK`: Downloads onto disk via a local program
  - `SYSTEM, BITS, BITSADMIN, CERTUTIL`

`ISESteroids` can apply character, number, or binary encoding obfuscation to parameters, variables, functions and strings. It can also remove comments and blank lines and apply unique id obfuscation to the script id. `ISESteroids` is a native program which includes a graphical tool for creating its obfuscations making the individual commands difficult to run automatically [5].

2.2 Deep Learning Methods

Many methods have been borrowed from static analysis of scripts as well as other areas of machine learning and applied to this problem. Victor Fang [1] describes FireEye, an NLP system for interpreting a PowerShell script by stemming commands. FireEye first decodes and tokenizes the script before stemming tokens to their semantic base. The decoding step allows FireEye to handle remote download and executable malware created by cradle and launcher obfuscation techniques. Decoded tokens are then used to create a feature vector for the script which is classified with a supervised algorithm such as Kernel SVM.

Hendler et al. [7] describes a method for detecting malicious PowerShell using computer vision and NLP techniques. The computer vision techniques encode the first 1024 characters of a Powershell command as a matrix where each row is a one hot vector with zero entries except for the code of the character at that index and apply a CNN to the matrices. The NLP techniques encode the command as a vector

of length 1024 containing the code for each of the first 1024 characters and feed this vector into an RNN. Both of these techniques focus on the character content of the script which as we have discussed can be easily manipulated through obfuscation.

Mou et al. [2] discusses a coding criterion based on AST representations of programs for use in deep learning. The criterion is used to produce a vector representation of the program that can be fed into a deep learning algorithm. The criterion uses AST node granularity because the finite number of node types makes learning feasible. A single neural layer codes each AST node using the representations of its children thereby learning a vector representation of a program from its structure.

Rusak et al.[8] found that families of malicious PowerShell can be classified fairly accurately using AST analysis simply for node count and depth. It further proposed a system for learning vectorized AST representations for more robust classification, including a distance measure for AST subtrees using the similarity between different node types. These same features may also be useful for associating obfuscated scripts with their original parent script.

2.3 De-Obfuscation

Some work has also been done to apply de-obfuscation to PowerShell before malware detection analysis [9]. PowerDrive implements a process which requires analyzing the script for layers of obfuscations before performing layer by layer de-obfuscation. Before de-obfuscation the script is cleaned to remove syntax errors or debugging commands before the proper de-obfuscation is performed. De-obfuscation can mean using a regex to find and remove common obfuscation patterns, such as concatenation in string obfuscation. It can also mean running an encoded script using `Invoke-Expression` and intercepting the decoded text of the script at run-time. Once a layer has been de-obfuscated the script is run to ensure that no errors were made to corrupt its execution. This is a problematic approach when dealing with malware because running the de-obfuscated script can potentially infect the system.

We can solve this by running the script in a virtual machine sandbox, but this requires extra steps to setup and partially defeats the purpose of malware detection because the scripts must still be run.

We employ obfuscation tools to create a dataset of files which are known to be obfuscated examples of particular scripts. We apply knowledge gained by exploring AST representations of PowerShell to classify malware families to our new goal of attributing obfuscations to their originating script. Finally, we consider the application of decoding tools to expand our ability to classify additional types of obfuscation.

Chapter 3

Dataset

We begin with a large corpus of PowerShell from Palo Alto Networks [11]. The dataset consists of a 3.76GB folder of 412,075 PowerShell scripts of unknown risk and a smaller set of 4,079 known malicious PowerShell scripts labeled with family types. In analyzing our dataset we discover that we have a large range of file sizes, shown in Figure 3-1. However, while our largest file is 18.9MB, our average file size is .01MB and most of our files are under 1MB. Since large files represent a small subset of our data, approximately 200 or .04% of the data, and some of our obfuscation tools do not obfuscate beyond a certain number of characters, we choose to exclude larger files. We exclude these files because we need to use these obfuscation tools with character limits and we expect that large files are also less likely to be used by attackers due to the limitations of these available tools.

3.1 Data Augmentation

We augment a subset of our dataset by obfuscating the scripts using freely available tools. We automate the process to obfuscate the scripts first using a single obfuscation type and then subsequently obfuscate the output scripts. This both mimicks techniques used by attackers to bypass malicious PowerShell detectors and ensures that we observe all possible combinations of obfuscation types. We use a subset of the data due to time constraints and because we expect that the effect of obfuscation

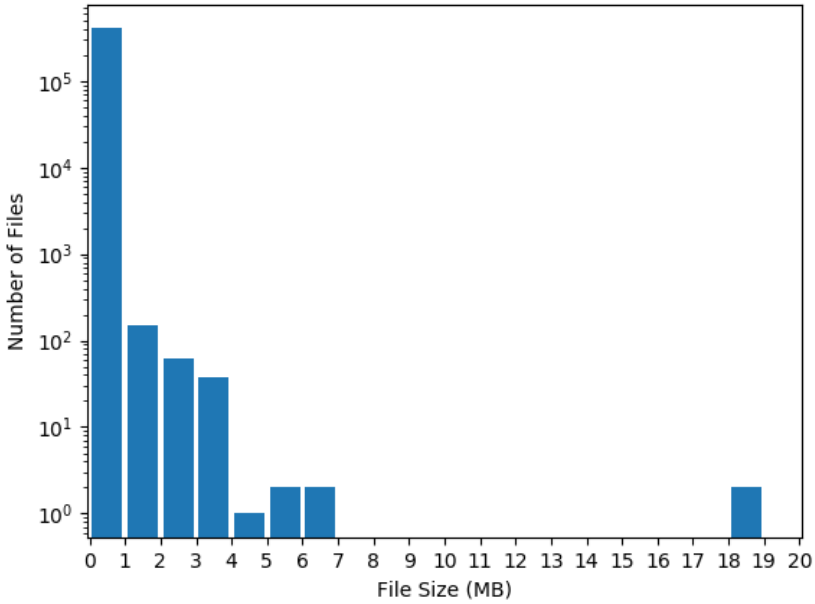


Figure 3-1: Dataset File Size: Most files are under 1MB with a small number ranging up to 19 MB

is similar on all files. We store our augmented data remotely since the number of obfuscations grows exponentially with each iteration of obfuscation applied. We apply each obfuscation once and then obfuscate each of those files again with the exception of `LAUNCHER`, `ENCODING` and `COMPRESS` obfuscations. We exclude these because they all involve obfuscation methods which must be de-obfuscated before extracting AST, as we describe in 4.1. We are left with 33 obfuscation types listed in Table 3.2 which are each applied once to the original files and once again to each obfuscated file. In total we obfuscate ten original files and create 1,122 obfuscations, 33 single type obfuscations and 1,089 chained obfuscations, for each of the ten original files. This gives us a total augmented dataset of 11,220 files in our augmented dataset. The relative sizes of our two datasets is summarized in Table 3.1

Table 3.1: Dataset Size: Number of Files in Full and Augmented Dataset [8]

Dataset	Data Folder	Number of Files
Full	Unknown Security	412,075
	Malicious	4,079
Augmented	Original Files (from the full dataset)	10
	Obfuscations Per File	1,122
	Total Obfuscated Files	11,220
	Unrelated or Other files (includes original files)	12

Table 3.2: Obfuscation Types: All Obfuscation Commands Applied [3]

Obfuscation Type	Token Type	Obfuscation Method
TOKEN	STRING	CONCATENATE
		REORDER
	COMMAND	TICK
		CONCATENATE
		REORDER
	ARGUMENT	RANDOM CASE
		TICKS
		CONCATENATE
		REORDER
	MEMBER	RANDOM CASE
		TICKS
		CONCATENATE
		REORDER
	VARIABLE	RANDOM CASE
	TYPE	CONCATENATE
		REORDER
	COMMENT	REMOVE
	WHITESPACE	RANDOM
	ALL	ALL
AST	NamedAttributeArgumentAst	REORDER
	ParamBlockAst	
	ScriptBlockAst	
	AttributeAst	
	BinaryExpressionAst	
	HashtableAst	
	CommandAst	
	AssignmentStatementAst	
	TypeExpressionAst	
	TypeConstraintAst	
	ALL	
STRING	CONCATENATE	
	REORDER	
	REVERSE	

Chapter 4

Methods

We began by exploring obfuscation techniques for PowerShell scripts and their effect on the look and structure of the code. We explore representations of these scripts that are robust to many types of obfuscation. We select features which contribute to these representations and automate their extraction. Finally, we use these features to explore and learn simple models for attributing obfuscated PowerShell to the original script in order to analyze their effectiveness.

4.1 Obfuscation

We implement automated obfuscation in Python with the following process, using one of the tools mentioned in Section 2, `Invoke-Obfuscation`. `Invoke-CradleCrafter` is not included because it creates a separate command which either launches or downloads the desired script. This means that we would have to recover the original script before extracting an AST, likely resulting in an identical AST. `ISESteroids` is not included since it uses a GUI to select individual commands to run obfuscation and therefore cannot be easily automated.

```
def obfuscate_random(dataset):
    # Choose ten random file in the dataset
    original_files = random.choice(dataset, 10)

    #Iterate through the files
    for original_file in original_files:

        # Obfuscate with each available command and write obfuscated
                                            scripts to new files
        for obfuscation_command in obfuscation_tools:
            # Obfuscated output filename will include original name
                                            and type of
                                            obfuscation
            new_obfuscated_filename = original_file.name +
                                            obfuscation_type + ".
                                            ps1"

            # A command to open powershell, run obfuscation and
                                            write the result to a
                                            file
            cmd = "powershell.exe -command obfuscation_command -
                                            script original_file -
                                            out
                                            new_obfuscated_filename
                                            "

            # Open a subprocess to run the command, write logging to
                                             stdout
            process = subprocess.Popen(cmd, stdout=sys.stdout)
            process.wait()
```

We first select ten random files from our dataset. We systematically apply each available obfuscation by deploying a Python process to run a PowerShell script which iterates through all available obfuscation commands in `Invoke-Obfuscation`. Finally, we write the obfuscated command to a new file whose name includes the original file

name and the type of obfuscation applied. In this way we can easily compare the same obfuscation methods across files as well as different obfuscation methods for the same file.

For our analysis we focus on three major types of obfuscation, token obfuscations, string obfuscations and Abstract Syntax Tree obfuscations. Each overarching category includes sub-types which refer to the specific item being obfuscated (`STRING, COMMAND, VARIABLE, WHITESPACE, AST node type`) or mode of obfuscation (`CONCATENATE, RANDOMCASE, REVERSE, RENAME, REORDER`) as described in Table 3.2. Launcher, compress and encoding obfuscations are excluded because they involve converting the command into a format that must first be decoded into the original script before its AST can be extracted. `ENCODING` obfuscations convert the script into another character format which the PowerShell AST extractor does not handle. `COMPRESS` obfuscation forces the code into a single line and applies compression which may keep us from extracting a full AST. `LAUNCHER` obfuscations are similar to the techniques implemented in `Invoke-CradleCrafter` which create a new command to launch the original script rather than obfuscating the script itself.

Our PowerShell for creating obfuscations takes the following steps:

1. Given a file name extract the file text as `$OriginalScriptBlock`.
2. Create nested arrays defining the available commands in Table 3.2
3. Loop through the arrays and concatenate each string with \ to create `$InvCommand`, for example: `TOKEN\STRING\1` which performs a token obfuscation on all strings by concatentation.
4. Call `Invoke-Obfuscation` with the extracted `$OriginalScriptBlock` and our created command `$InvCommand`, in the form:
   `Invoke-Obfuscation -ScriptBlock $OriginalScriptBlock -Command $InvCommand -Quiet`
5. Write the output obfuscation to `$OutputFilePath`

Once we have applied each obfuscation type to the original file we can iterate through the new obfuscated files and apply the same obfuscations again to obtain files with multiple types of obfuscation. This is a common technique used by attackers to increase the likelihood that their script will break signature detection. We can do this many times in order to explore whether the impact of obfuscation decreases, increases or remains constant with each iteration.

4.2 AST Extraction

As we create each obfuscation we also extract its AST. An AST describes the control sequence of a program and includes nodes for functional blocks in PowerShell, such as a variable assignment or function call. In PowerShell there are a limited number of AST node types making AST based analysis more tractable than other platforms. To perform the extraction we import a python script from previous work done in ALFA Group [8]. The script parses a PowerShell script and writes a text file which lists each node type along with its parent and the lines it encompasses.

4.3 Visual Analysis

To begin our analysis we ensure that the proper obfuscation has been applied by examining a few files visually as shown in Figures 2-1 and 2-2. We expect to be able to see changes immediately for string obfuscations since this type of obfuscation changes the readability of the code. For AST obfuscations we can inspect the generated AST files and ensure that the order or type of AST node varies in some places. We do not expect the length of the AST file to change very much in either case since these obfuscations simply modify strings or change the order of AST nodes. These methods for visual observation will also inform our later distance metric exploration since we expect certain changes in the file for certain obfuscation types.

4.4 Distance Measures

Once we have obfuscated scripts we can use them to calculate distance measures. We begin with static analysis techniques by calculating cosine distance between the distribution of characters in the original and obfuscated files. We then calculate a similar cosine distance between the distribution of AST node types in both files. In addition to cosine distance we calculate a hamming distance between the AST and also extract bigrams of child and parent node types from the AST to calculate cosine and hamming distances.

4.4.1 Cosine Distance

We begin by calculating cosine distance between character and AST node type distributions for our obfuscated files versus their original file. We use spatial cosine distance from SciPy.org for the calculation which calculates $1 - cosine\_similarity$.

Character Count

We iterate through each character in our original script and create a dictionary where the keys represent the characters in the script and the value is the number of times it appears in the script. We do the same for the obfuscated script. Once both dictionaries are created we create a vocabulary by combining the two sets of keys. We then iterate through the vocabulary and create two vectors with equal length to the vocabulary where each value is the number of times that character appeared in the script. We calculate the cosine distance between these two vectors as our distance metric.

```
# Create each vocabulary
original_vocab = {}
obfuscated_vocab = {}

# Iterate through the characters of the original script
for c in original_script.get_characters():
    original_vocab[c] += 1

# Iterate through the characters of the obfuscated script
for c in obfuscated_script.get_characters():
    obfuscated_vocab[c] += 1

# Create a combined vocabulary
vocab = set(obfuscated_vocab.keys() + original_vocab.keys())

# Create vectors for each script
original_vec = np.zeros(len(vocab))
obfuscate_vec = np.zeros(len(vocab))

# Iterate through the vocabulary and assign vector values
for i, v in enumerate(vocab):
    original_vec[i] = vocab[v]
    obfuscate_vec[i] = vocab[v]

# Calculate the cosine distance
cos = spatial.distance.cosine(original_vec, obfuscate_vec)
```

AST Node Type

We use a similar process in calculating cosine distance for AST node type except that we iterate through the nodes in the AST text file rather than the characters of the script. Therefore our vocabulary keys are AST node types and the values are the number of times that node type appears in the AST.

4.4.2 Hamming Distance

```
# Create empty arrays
original_nodes = []
obfuscated_nodes = []

# Iterate through the original AST file
for node in original_ast_file.get_nodes():
    original_nodes.append(node)

# Iterate through the obfuscated AST file
for node in obfuscated_ast_file.get_nodes():
    obfuscated_nodes.append(node)

# Get the minimum length for truncation and maximum length for
                                    scaling
minlength = min(len(original_order), len(obfuscate_order))
maxlength = max(len(original_order), len(obfuscate_order))

# Get an array that is True where the two arrays differ
differences = np.array(original_order[:minlength]) != np.array(
                                    obfuscate_order[:minlength])

# Calculate Hamming distance
ham = (np.sum(differences) + (maxlength - minlength))/float(
                                    maxlength)
```

We iterate through the AST text file and create two arrays which are an ordered list of the node types present in the file. We sum the number of locations at which the two arrays differ. We handle different length AST by truncating both lists to the minimum length and adding the difference between the two lengths to the sum of differing node types. We then divide by the maximum length in order to scale the distance relative to AST size.

4.4.3 Bigram Distances

The AST text files generated list each child node along with its parent. We parse the text files, treating each of these parent child pairs as a single node, and apply the same distance metrics from earlier sections.

Cosine

In this case the cosine distance is calculated from the distribution of parent child node pairs in the AST. We create a dictionary where the keys are strings containing the two node types separated by a comma and the values are the number of times that parent and child combination appeared. We then proceed with the same method as Section 4.4.1 using this modified vocabulary

Hamming

We calculate bigram hamming distance just as in Section 4.4.2 except that the value at each index in the node list is now a parent child pair.

4.5 Extracting Features and Labels

In order to evaluate our distance metrics we employ them as features in a model that classifies obfuscated files to their original file. We have stored the AST in folders bearing the name of their original file. This will serve as the labeling scheme for our dataset. We extract the leaf directory name and assign it a class number which becomes the label for each file within the directory, resulting in ten labels $\{0 - 9\}$. We split the data into train and test sets by shuffling the dataset and selecting the last twenty percent of it as test files. We use this approach so that we have balanced classes in both our train and test sets. We use the same train and test sets for all of our modeling experiments

4.5.1 Distances to Original Files

Our features are the obfuscated file's distance to each original file in our augmented dataset. We calculate four feature vectors, one for each of our distance metrics, of length ten for each obfuscated file in our augmented dataset. Each index of the feature vectors holds the value of the distance metric for the obfuscated file compared to the original file whose label corresponds to that index.

4.5.2 AST Structure

We draw upon work done in Rusak et al. [8] to extract two additional features from our AST. We parse the AST to calculate the depth and total node count of the tree. We use these two features alone in baseline experiments and include them with our distance features in later experiments to determine if our features are complimentary.

4.6 Modeling

In order to evaluate the usefulness of our features and distance measures we use them to learn simple models. We test three types of models from Scikit-learn: k-Nearest Neighbor classification, Random Forest classification, and Logistic Regression. We train each model on our AST structure features alone as well as each distance type feature vector separately and all features together.

We experiment with three models. Once again drawing upon previous work [8], we experiment with a Random Forest classifier with a maximum depth of 11 and a random initialization. We also experiment with a simple k-Nearest Neighbor algorithm which assigns labels based on the nearest five previously seen examples. Finally, we apply a Logistic Regression model with no regularization.

Chapter 5

Experiments and Results

5.1 Experimental Setup

We design our experiments to quantitatively explore the effect of obfuscations on a PowerShell script. We would like to compare our obfuscated and non-obfuscated PowerShell quantitatively and determine how we can represent scripts such that an obfuscated script can be attributed to its non-obfuscated counterpart. To this end we compare obfuscations to their original scripts using several distance measures and separately compare non-obfuscated scripts to each other using the same measures. Finally, we use these distance metrics and features to experiment with models and determine if our representations of PowerShell can accurately measure difference in a way that is useful for detection.

5.1.1 Augmenting Data

As described in section 4.1 we select ten random files from our dataset to create an augmented dataset of obfuscated files. We create obfuscation files for each type of obfuscation we are interested in. We have a total of 33 obfuscation commands between TOKEN, STRING, and AST command which results in 1,122 obfuscated file per original file, 33 single obfuscations and 1,089 files with two obfuscations applied in series as described in Table 3.2. We use a PowerShell script provided in Rusak

et al. [8] to extract AST from each obfuscation. This script requires a Windows environment which we access through the CSAIL OpenStack experimental windows 2012 image. This also allows us to access the original dataset and store large volumes of generated obfuscations and AST remotely.

5.1.2 Exploring File Distances

We evaluate our four distance metrics by comparing the distance between an original and an obfuscated file to the distance between unrelated files. We plot several histograms for each file showing the distribution of each distance metric for three comparisons: all obfuscations of the file, the second obfuscation applied versus the first obfuscation applied, and all obfuscations of the file versus all other original files. We also split these histograms into only AST obfuscations and only string based obfuscations in order to compare the performance of different metrics for different types of obfuscation.

5.1.3 Modeling

We extract feature vectors for each of our original and obfuscated files for each distance metric. We assign each original file with an integer class label {0-9} and label our feature vectors with the number corresponding to each file's original file. We split the data into 80% train and 20% test and apply the same split to the feature vectors for each distance metric.

We experiment with three different models. The first is Random Forest Classification in order to compare our results to previous work done in Rusak et al. [8]. The second is k-Nearest Neighbor classification, because the algorithm assigns new examples to groups of previously classified similar examples which is identical to the process we would like to apply to our problem. Finally, we experiment with Logistic Regression classification as a baseline. We train each model with our training vectors for each distance metric and compare the accuracy for each model across metrics.

5.1.4 Testing Obfuscations

Once we have obfuscated our scripts, it is important to test our obfuscated code to ensure that the functionality is preserved. To do this safely we can run our scripts within a sandbox and check that the malicious payload is delivered. In addition, we manually inspect files which are outliers in our data exploration. We use this inspection to determine what is different about these files and inform our future data and distance metric exploration.

Sandbox Setup

We set up Virtual Box on an Ubuntu 18.04LTS machine that we disconnect from the network. We load a Windows 10 image into a new virtual machine in Virtual Box. In order to run the necessary scripts we move them out into our repository which we have pulled onto the virtual machine. We do not have storage constraints in this step since we only need to test a small number of files, one for each obfuscation type and a number of outliers.

Determining Parameters

In order to run these scripts we must determine their dependencies as well. We select one of our random scripts which does not need parameters to run each obfuscation type for easy testing. For our outliers we examine the original script to determine the necessary parameters and use the same parameters to run the obfuscated outliers.

5.2 Results and Discussion

5.2.1 Distance Metrics

Cosine Distance

We start with a cosine distance metric for character distribution because it uses no information from the structure of the script and provides a good baseline for us to

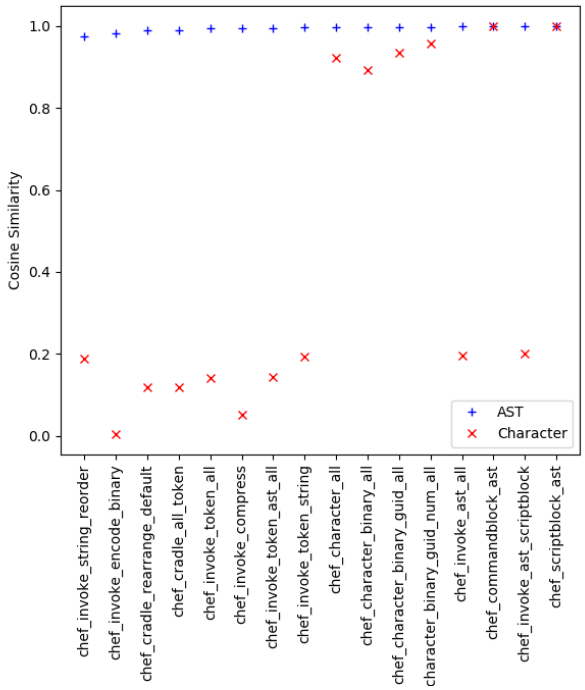


Figure 5-1: Scatterplot of Obfuscated Script Character and AST Similarity with Original: AST similarity is high while character similarity is quite low and more variable.

improve upon. Our first improvement is to incorporate the structural information contained in AST by computing a cosine distance for AST distribution. Since we are not analyzing the order of AST nodes with this metric we are still including only a minimal amount of structural information. We compare cosine distance measures for character distribution and AST node type distribution by plotting their distance scores across several obfuscation techniques for a single file in Figure 5-1. This plot suggests that AST based measures may be the most useful in determining script similarity and that we should explore other AST based analysis.

Character Count

As expected character obfuscations can be shown to impact character based similarity measures more strongly. Additionally, the effect of some obfuscation techniques is certainly more pronounced than others.

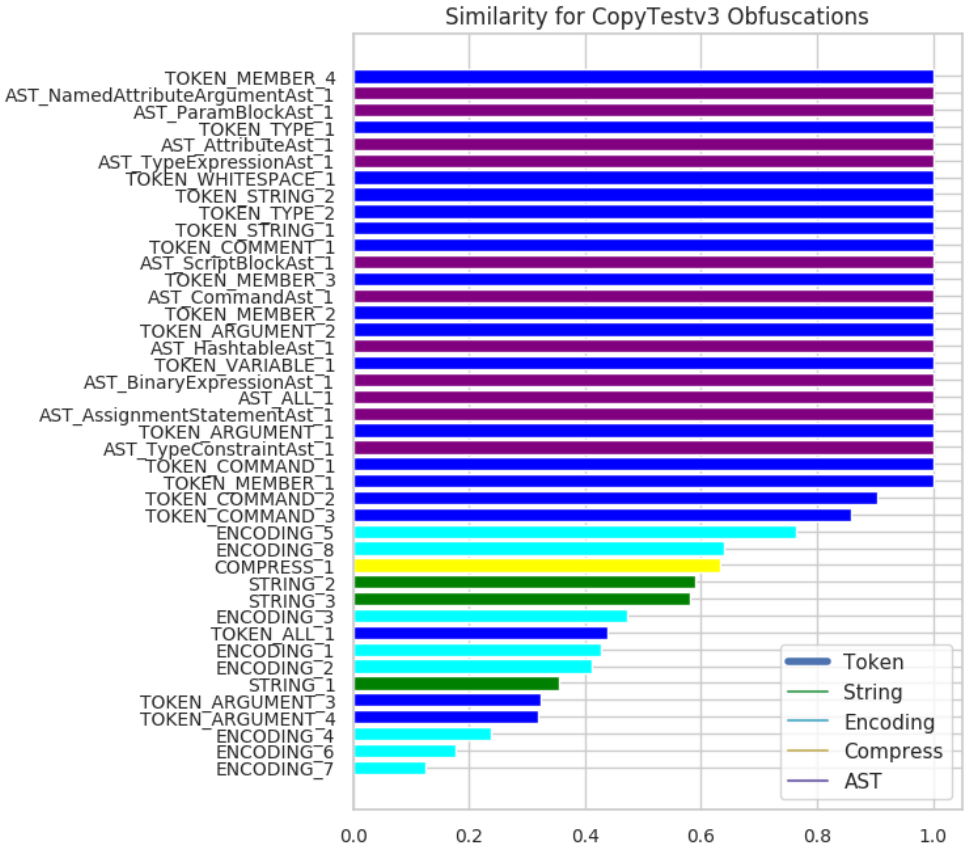


Figure 5-2: Histogram of Obfuscated Script AST Similarity with Original: cosine similarity is one for all AST files but varies for string based obfuscations.

AST Node Type

AST node count based cosine similarity is relatively stable across all techniques in Figure 5-1. However, the AST based metric seems to be impacted to a greater degree by character based obfuscation as shown in Figure 5-2. These cases may be an example of a truncated script or an obfuscation which has altered the function of the code. In addition, the encoding and compress obfuscations have low similarity. We know that this is likely because these obfuscations make recovery of the AST impossible without first de-obfuscating the script.

Next, we want to determine how effective our cosine distance metric between AST distributions might be for distinguishing between an obfuscated file created from the

original file and an unrelated file. To this end we plot a histogram of the cosine distance of obfuscations to the original file against the cosine distance of the same file to other original files in Figure 5-3. This figure shows that while there is some overlap most obfuscated files will have a lower distance from the original file than an unrelated file. This trend suggests that we can threshold the cosine distance of a file from an original script to classify the script as an obfuscated version of the original. However, as shown in Figure 5-4, the threshold between an obfuscated script cosine distance and an unrelated script cosine distance can vary based on the original file we choose, making a single thresholding value inaccurate for multiple scripts. We can also see this overlap in Figure 5-5 which shows the aggregated histogram of obfuscated file distance versus unrelated file distance for all files. There is a lot of overlap of obfuscated file distances with unrelated file distances throughout the range of possible distances. If we examine Figure 5-6, this overlap is unsurprising. The figure shows the cosine distance for all original files against all others and the majority of the distances are very small, so we can conclude that cosine distance does not distinguish between unrelated scripts very well.

We also plot the cosine distance histogram for AST only obfuscations versus character based obfuscations to determine if this metric might work better for one type of obfuscation over another. As we can see in Figure 5-7 this seems to be partially true. Character based obfuscations have higher distance from the original in general. We might not expect this since character obfuscations should not impact the AST structure but this could be a result of corrupting the script through obfuscation. There is also a large spike in character obfuscations which have very low distance from the original so we would need to explore those further to determine why they do not follow the pattern.

Hamming Distance

We continue to explore additional structural information of the script by computing Hamming distance between the list of nodes in the AST. The hamming distance is calculated by computing the percentage of equal indexes from an ordered list of nodes

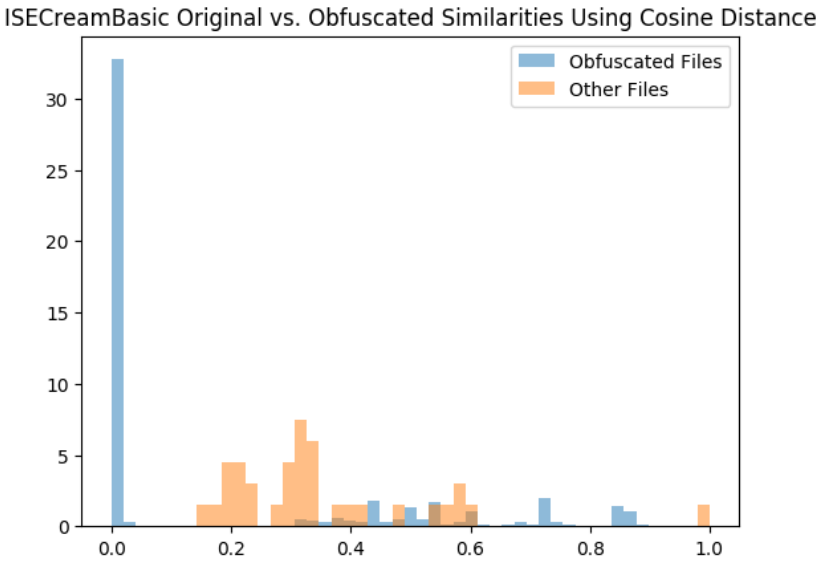


Figure 5-3: ISECreamBasic Cosine Distance Histogram for Obfuscated Scripts vs. Unrelated Scripts: It is clear that obfuscated scripts have a lower distance score than unrelated scripts with some exceptions.

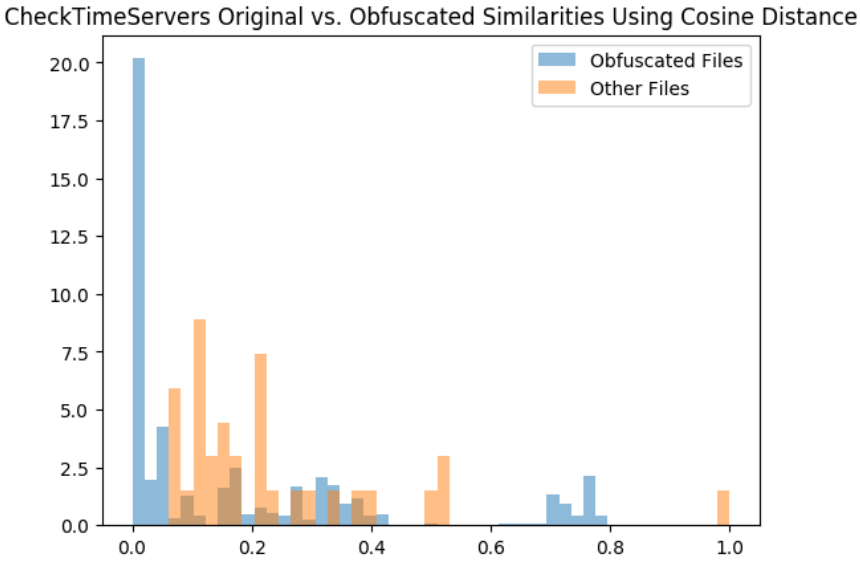


Figure 5-4: CheckTimeServers Cosine Distance Histogram for Obfuscated Scripts vs. Unrelated Scripts: The threshold at which it is likely a script is unrelated to the original is much lower here.

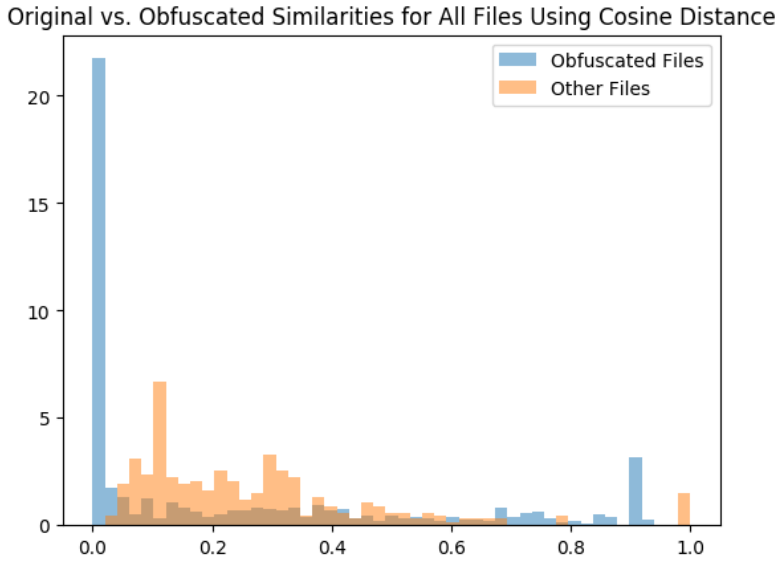


Figure 5-5: Cosine Distance Histogram for All Obfuscated scripts vs. All Original scripts: The two histograms overlap throughout most of the range of distances.

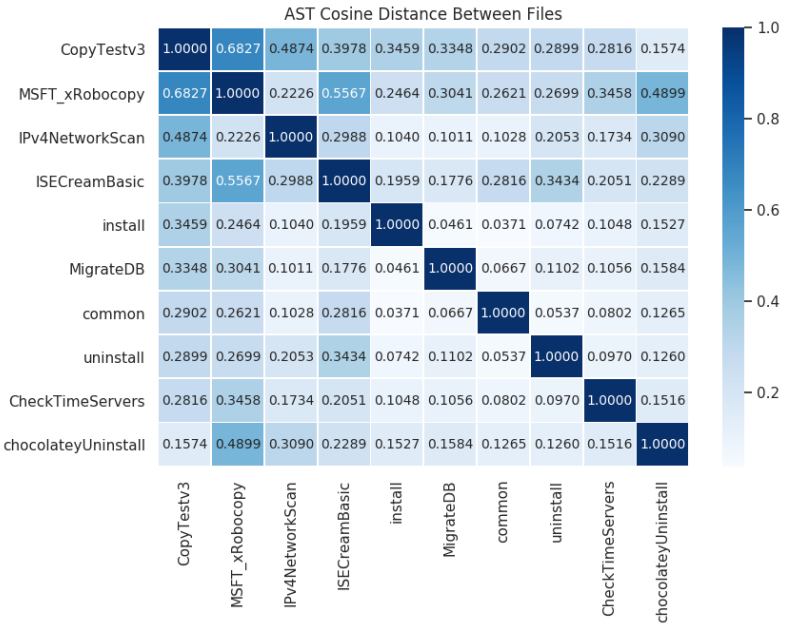


Figure 5-6: Cosine Distance Heatmap for All Ten Original Files: Most distances are very low with the exception of a few in mid range.

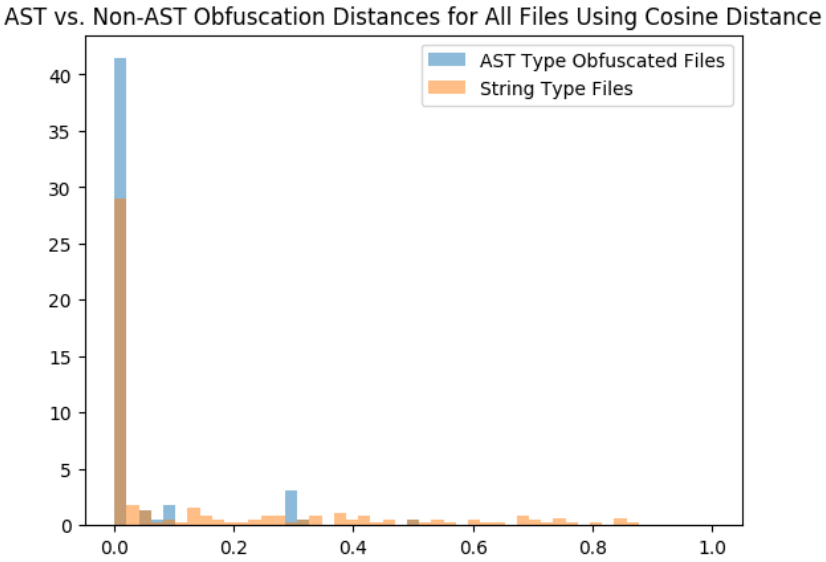


Figure 5-7: Cosine Distance Histogram for AST Only Obfuscations vs. Character Only Obfuscations For All Obfuscated Files: There is a large spike at low distance for both types but character based distance is higher for the remaining examples.

in the AST. This method incorporates information about when an AST node appears in the script into our distance metric. Figure 5-8 shows the histogram of obfuscated file hamming distance from the original versus unrelated file hamming distance. Here we see much higher distance values than we did for cosine distance for both obfuscated and unrelated files. We expect this because the hamming distance incorporates more structural information about the script. We also see a slightly more pronounced difference between the distance for an unrelated file and an obfuscated file when we plot this comparison for all files in Figure 5-9. As we confirm in Figure 5-10, most unrelated files have a distance of 0.85 or greater from the original file while obfuscated files have distances over the entire range. Even though there is still significant overlap we expect that hamming distance will be a more predictive feature for our attribution models.

We again compare hamming distance for AST and character based obfuscations in Figure 5-11. Here we continue to see a spike for both types at low distances. However, this time AST obfuscations show very high distance values along with the character

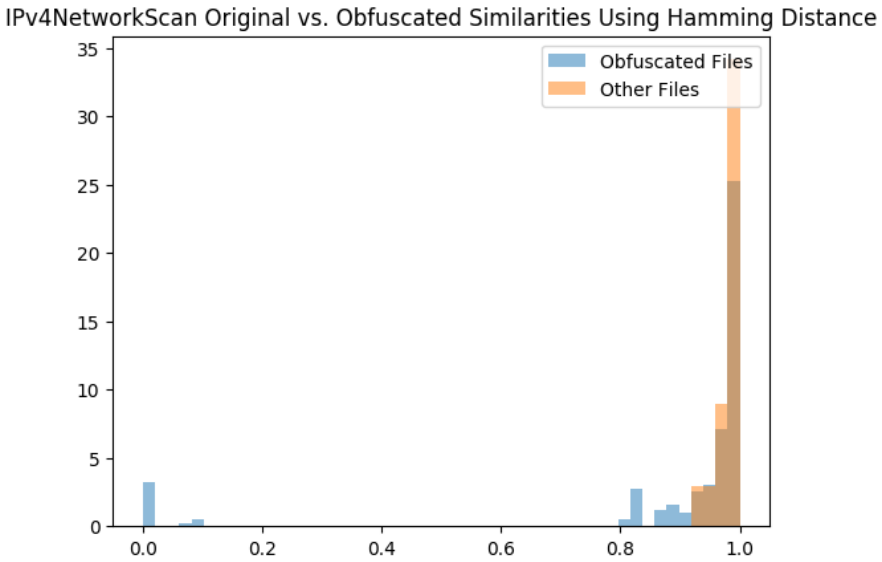


Figure 5-8: IPV4NetworkScan Hamming Distance Histogram for Obfuscated Scripts vs. Unrelated Scripts: While there is overlap between the histograms at high distances, unrelated files do not have distances less than 0.85.

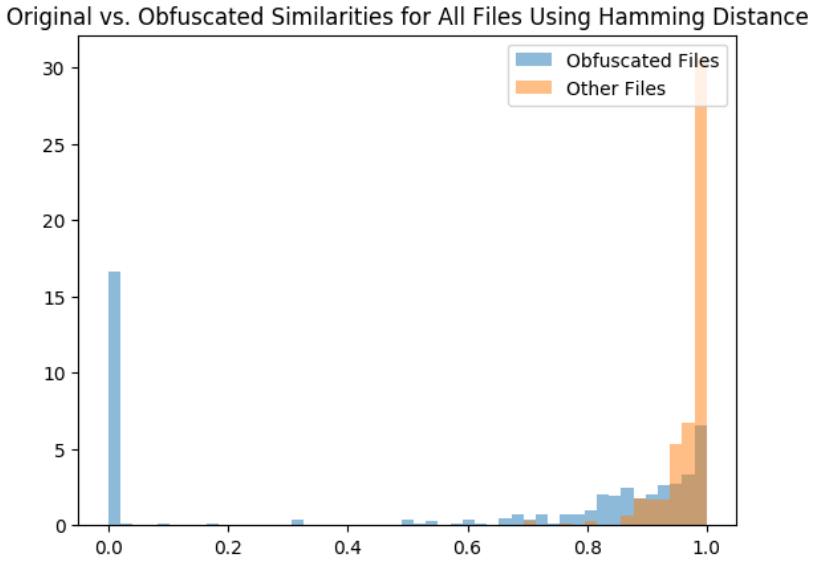


Figure 5-9: Hamming Distance Histogram for Obfuscated Scripts vs. Unrelated Scripts for All Ten Original Files: Unrelated files distances are clustered near 1 while obfuscated file distances are spread over the entire range.

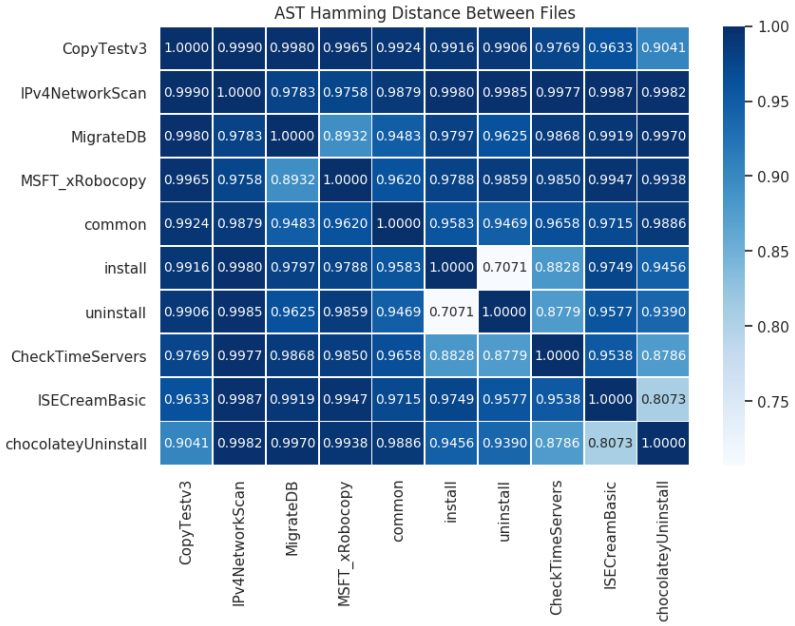


Figure 5-10: Hamming Distance Heatmap for All Ten Original Files: All distances are over 0.7.

obfuscations. We expect this for AST obfuscations we are now considering AST node order within the script and AST obfuscation is likely to change the order of nodes. This is less intuitive for character obfuscations since there should be no impact on the type of node however it may be a result of script corruption. There is not much difference in the distances between the two types for this metric so it is not likely to perform better for one than the other.

Bigram Distances

Since incorporating structural information of the script improved our ability to distinguish between obfuscated and unrelated files, we incorporate more structural information by leveraging the available parent child relationships in the AST. The AST files list parent and child nodes in pairs so we can easily treat these bigram pairs as single nodes when parsing the AST. We expect this to result in higher distances because there are n^2 possible node types now where in the previous calculation there were only the number of PowerShell node types, n.

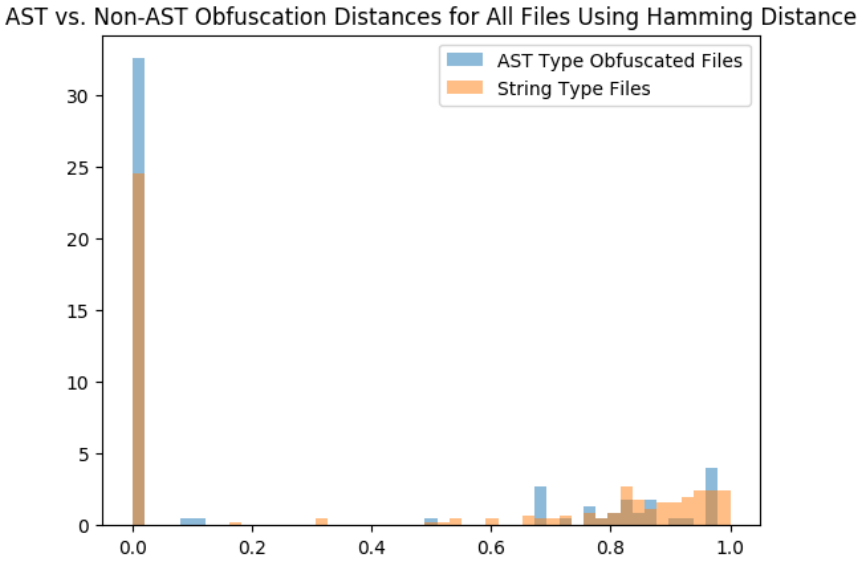


Figure 5-11: Hamming Distance Histogram for AST Only Obfuscations vs. Character Only Obfuscations For All Obfuscated Files: Both types include a spike at low distance with most of the distribution at high distance.

Cosine Once again we examine the distributions of cosine distance for obfuscated files versus unrelated files. In Figure 5-12 we see that using bigrams improves our ability to distinguish unrelated files from obfuscated ones using cosine distance. Obfuscated files have a large spike at low distance while unrelated files no longer include this spike, despite the fact that they still have low distance measures. Unfortunately, this trend is not seen in all files. Figure 5-13 shows the aggregated histograms for all ten original files which very similar to the plot in Figure 5-5. We do see a small shift toward higher distances for unrelated files when we examine the heatmap in Figure 5-14. We suspect that unrelated files have close distance measures as a result of the small number of AST nodes used in PowerShell, since this means that all PowerShell must use many of the same nodes.

Hamming We find that for bigram nodes, hamming distance of unrelated files is even closer to one than for unigram nodes, as shown in Figure 5-15. However, the distances for obfuscated files also increase. We expect this because there are a larger number of possible parent and child node combinations and therefore it is more likely

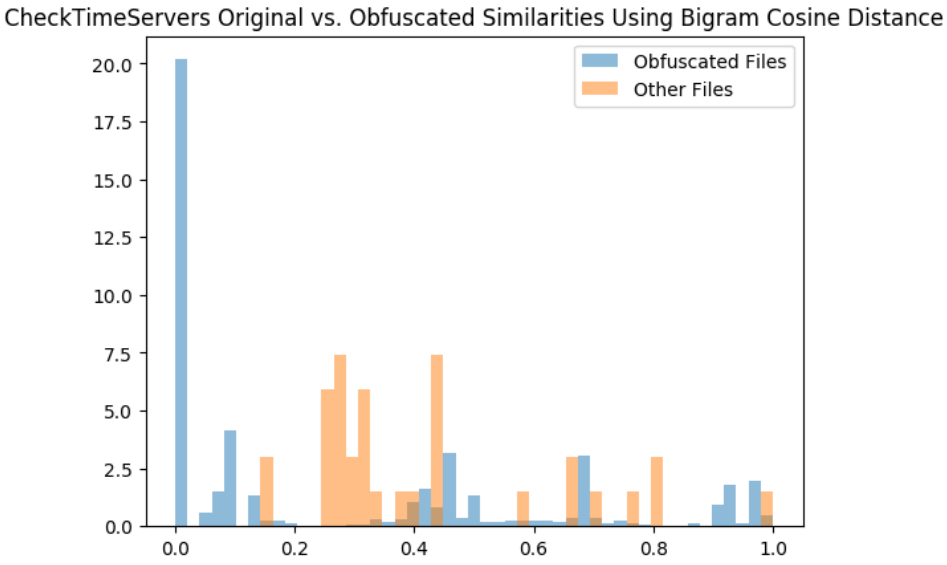


Figure 5-12: CheckTimeServers Bigram Cosine Distance Histogram for Obfuscated Scripts vs. Unrelated Scripts: Obfuscated scripts show a spike at low distance while unrelated scripts are distributed over all distances under 0.5.

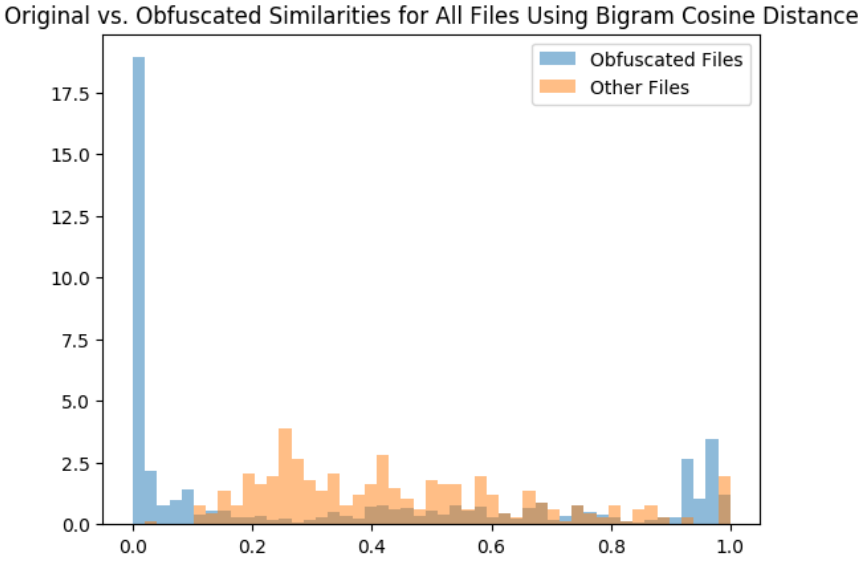


Figure 5-13: Bigram Cosine Distance Histogram for Obfuscated Scripts vs. Unrelated Scripts for All Ten Original Files: The two histograms overlap throughout most of the range of distances with more volume of unrelated file distance at lower distances.

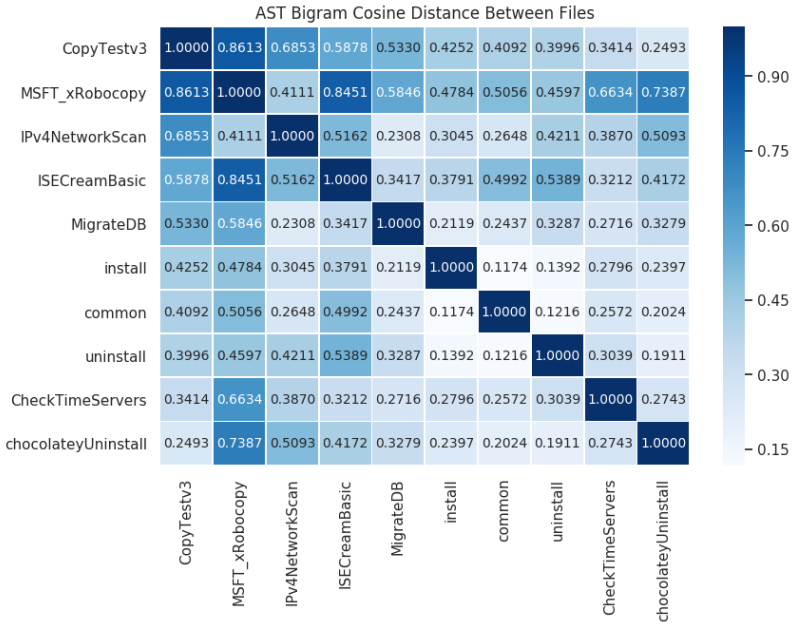


Figure 5-14: Bigram Cosine Distance Heatmap for All Ten Original Files: Most distances are still very low but all have increased somewhat.

for indexes not to match. We see the same trend in the aggregate histograms shown in Figure 5-16. The distances for both unrelated and obfuscated files are higher than for unigram hamming distance. We confirm this trend by examining the heatmap for unrelated file distance in Figure 5-17. Overall this demonstrates that bigram hamming distance is best at detecting unrelated file distance but not necessarily obfuscated file distance.

Multiple Obfuscation Effects

We examine the difference in distance distribution between files which have been obfuscated once and files which have been obfuscated twice, in order to determine how multiple obfuscations impact our distance metrics. Figure 5-18 shows the distribution of distances for single type obfuscated files against that of double type obfuscated files for all distance metrics. In all cases the distance distribution is slightly higher for double obfuscation than single obfuscation. We do not expect this to have much impact on our cosine distance metrics given that unrelated files have varied distances

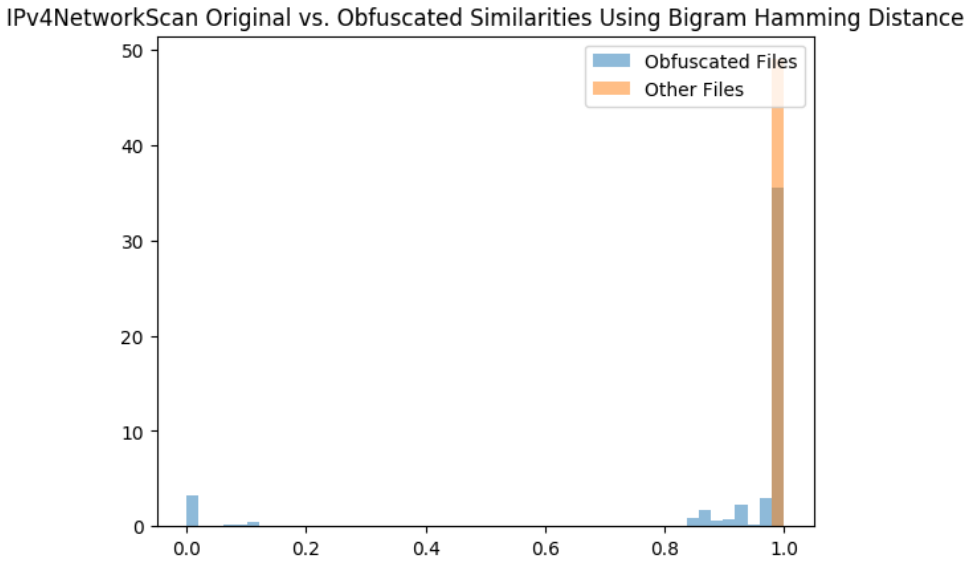


Figure 5-15: IPV4NetworkScan Bigram Hamming Distance Histogram for Obfuscated Scripts vs. Unrelated Scripts: All unrelated files have a distance over 0.99.

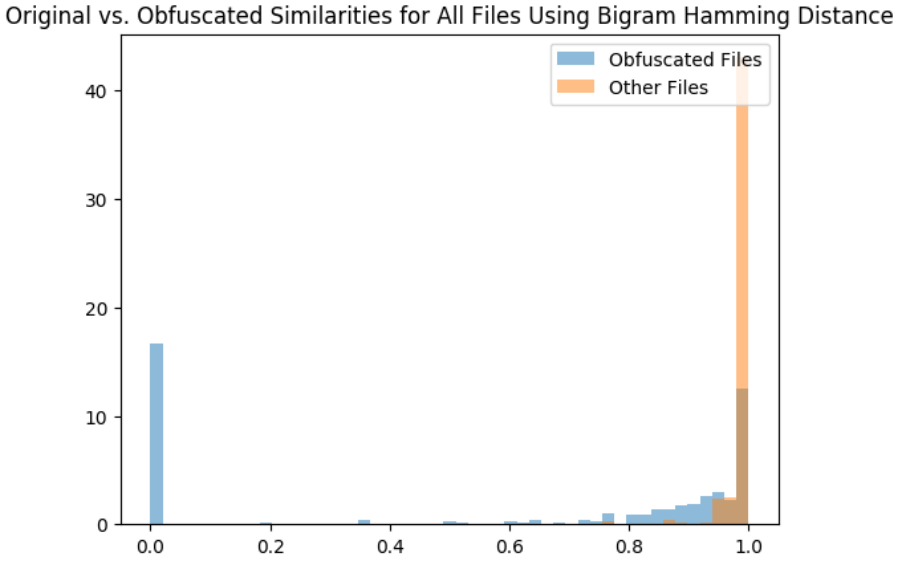


Figure 5-16: Bigram Hamming Distance Histogram for Obfuscated Scripts vs. Unrelated Scripts For All Ten Original Files: The plot is nearly identical to Figure 5-9 but with more volume toward the extreme ends of the distance range.

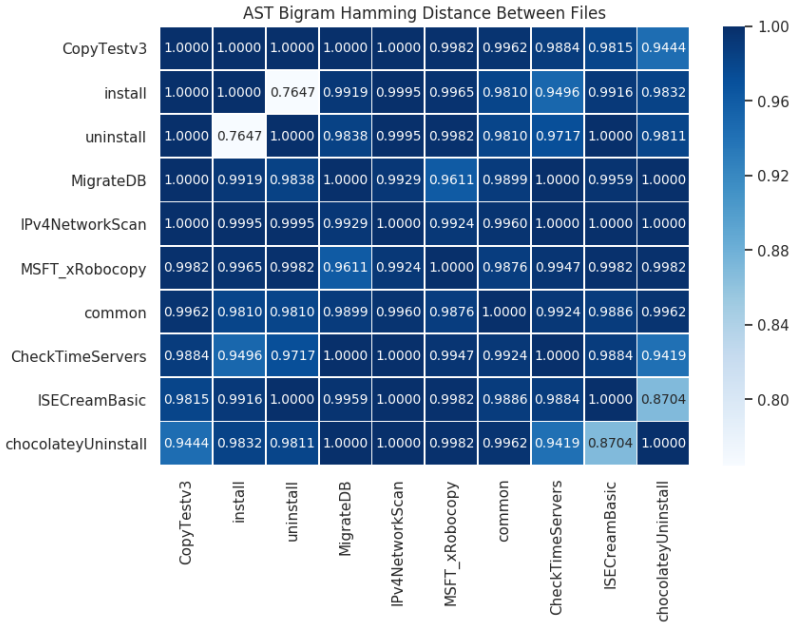


Figure 5-17: Bigram Hamming Distance Heatmap for All Ten Original Files: Distances are even higher than for unigram hamming distance suggesting that this is a more accurate metric.

for this metric. However, given that unrelated files have high distance values for hamming distance, this indicates that hamming distance metrics may become less useful for files in which multiple obfuscations have been applied. Their distance values may approach those of unrelated files after multiple obfuscations.

5.2.2 Features

We experiment with different combinations of features to determine the best approach to attributing obfuscations to their original files using our distance metrics. We train each of our classifiers on AST node count and depth alone, each of our distance metrics separately, and all of our features together. We hope to find a subset of features that is sufficient for our problem in order to keep our classifier simple. We compare the accuracy of each classifier when trained on each set of features in Table 5.1. We can see that including all features in model training results in higher accuracy for two of our models. We also see that our distance features perform better for k-Nearest

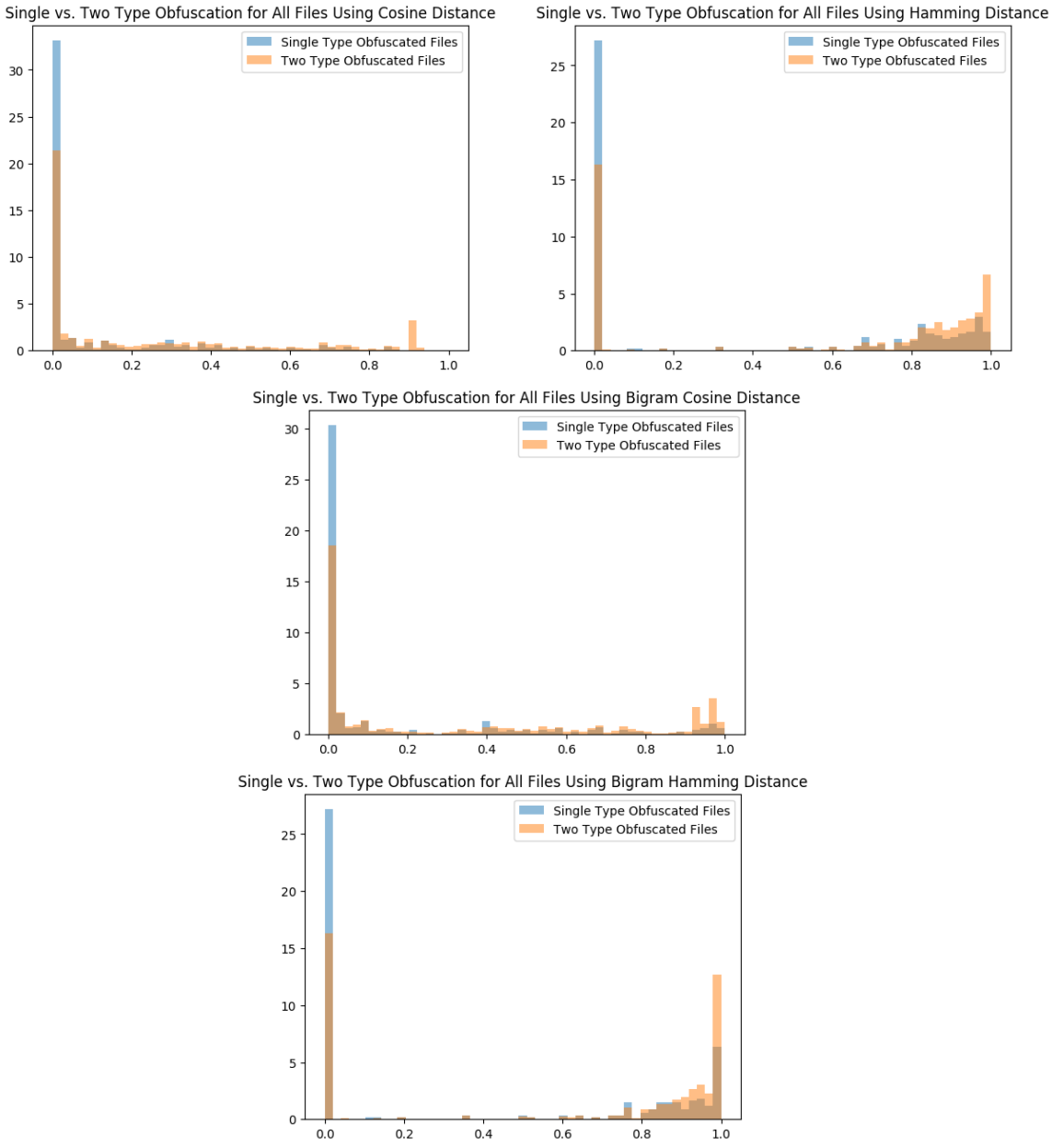


Figure 5-18: Distribution Histogram of Single Obfuscations vs. Double Obfuscations for All Distance Metrics: For each distance metric double obfuscations show distance concentrations at higher values.

Neighbor and Logistic Regression models than the baseline AST only features, node count and depth. In fact, using hamming distance features alone and a k-Nearest Neighbors model we can attribute obfuscated code to its original file with higher accuracy than any model using only AST node count and depth.

Table 5.1: Model Accuracies: Test accuracies of each classifier trained on a subset of features. [8]

Classifier	Features	Accuracy
Random Forest	AST features	.882
	Cosine distance	.732
	Hamming distance	.879
	Bigram Cosine distance	.748
	Bigram Hamming distance	.877
	All Features	.898
kNN	AST features	.864
	Cosine distance	.882
	Hamming distance	.888
	Bigram Cosine distance	.885
	Bigram Hamming distance	.877
	All Features	.871
Logistic Regression	AST features	.312
	Cosine distance	.724
	Hamming distance	.700
	Bigram Cosine distance	.778
	Bigram Hamming distance	.685
	All Features	.835

5.2.3 Models

As shown in Table 5.1 our Random Forest Classifier performs best out of all classifiers and our distance metrics contribute to a boost in performance over simple node count and depth of the AST. However, we discover that we can also achieve fairly high accuracy with a k-Nearest Neighbor classification algorithm trained on hamming distance features. This is promising in terms of improving clustering techniques for this problem. We consider clustering techniques particularly important here because the nature of the problems suggests that we are looking for an accurate grouping of obfuscated files to their original files.

Chapter 6

Conclusions and Future Work

We set out to understand the effect of obfuscation on PowerShell both qualitatively and quantitatively. We were able to visually identify different types of obfuscation and determine distance metrics for obfuscation both based on the characters in the file and the structure of the program. We examined these distance metrics as features for models to attribute obfuscated files to non-obfuscated files and determined that simple distance features based on AST structure are sufficient for achieving high accuracy. We discovered that the order of AST nodes is important for creating a useful representation of a script since our Hamming distance features led to more accurate attribution models. We also discovered that features of the entire AST tree, such as node count and depth, are important features for attribution models as well and that by combining distance features with AST tree features we can achieve higher accuracy than with either set of features alone.

Our results are promising but there are many areas for further exploration. As described in Section 2, research is ongoing in the area of decoding obfuscated PowerShell in order to better analyze its functionality and security risk. We would like to incorporate some of this work and expand the types of obfuscation that we examine to include the `ENCODING`, `COMPRESS`, and `LAUNCHER` techniques mentioned earlier. This would also allow us to include other obfuscation tools such as `Invoke-CradleCrafter` which focus on encoded and remote downloaded PowerShell.

To improve our features, we would like to include additional valuable information in the structure of AST subtrees which we have not explored in this work. We would like to extract subtree features such as depth and child count as well as the full AST features we currently use and experiment with whether these additional features might improve our models. We would also like to refine our features, perhaps by calculating distance measures using only less common AST nodes, since most scripts will use many of a small number of AST nodes. To test the robustness of our current features, we would like to explore hamming distance when all AST are of similar size. Specifically, we would like to test if unrelated file's hamming distances would be lower for AST of similar size. We would also like to test our distance metrics on more heavily obfuscated examples. We were able to show that multiple obfuscations increase both of our distance metrics but further research is needed to determine how much the distance increases and for how many obfuscations it continues to increase. Further exploration of outliers to our distance metrics is also needed. In preliminary analysis we were not able to draw solid conclusions about our outliers.

To improve our models we would like to experiment with parameter tuning of the k-Nearest Neighbor model as well as additional and refined feature sets. Our models were quite accurate on our relatively small augmented dataset but we would like to see if these models generalize to larger obfuscated and non-obfuscated datasets. We would also like to test our models against datasets that are known to be solely benign or malicious to see if they perform differently for specific conditions.

If these results are promising we would attempt to build malware detectors based on self-obfuscated augmented datasets of known malicious PowerShell. We would also explore whether detectors can be built using only benign files and their obfuscations.

Bibliography

[1] Victor Fang. 2018. Malicious PowerShell Detection via Machine Learning. https://www.fireeye.com/blog/threat-research/2018/07/malicious-powershell-detection-via-machine-learning.html.

[2] Lili Mou et al. 2015. Building Program Vector Representations for Deep Learning. In International Conference on Knowledge Science, Engineering and Management. Springer, 547âĂŞ553.

[3] Daniel Bohannon. 2018. Invoke-Obfuscation v1.8. https://github.com/danielbohannon/Invoke-Obfuscation.

[4] Daniel Bohannon. 2018. Invoke-CradleCrafter v1.1. https://github.com/danielbohannon/Invoke-CradleCrafter.

[5] Tobias Weltner. 2016. PowerShell Obfuscator. http://www.powertheshell.com/powershell-obfuscator/.

[6] Abdullah Al-Dujaili et al. 2018. Adversarial Deep Learning for Robust Detection of Binary Encoded Malware. In 2018 IEEE Security and Privacy Workshops (SPW). IEEE, 76âĂŞ82.

[7] Danny Hendler et al. 2018. Detecting Malicious PowerShell Commands using Deep Neural Networks. In Proceedings of the 2018 on Asia Conference on Computer and Communications Security. ACM, 187âĂŞ197.

[8] Gili Rusak et al. 2018. POSTER: AST-Based Deep Learning for Detecting Malicious PowerShell. In 2018 Conference on Computer and Communications Security (CCS). ACM, New York, NY, USA, 3 pages.

[9] Denis Ugarte et al. 2019. PowerDrive: Accurate De-Obfuscation and Analysis of PowerShell Malware. https://arxiv.org/pdf/1904.10270.pdf.

[10] JuanPablo Jofre et al. 2018. PowerShell Scripting | Microsoft Docs. https://docs.microsoft.com/en-us/powershell/scripting/overview?view=powershell-6.

[11] PowerShell Corpus. Palo Alto Networks. Fileset, 2018.