**FITEE**

# Generic, efficient, and effective deobfuscation and semantic-aware attack detection for PowerShell scripts[*][&]

Chunlin XIONG[†1], Zhenyuan LI[1], Yan CHEN[2], Tiantian ZHU[3],

Jian WANG[1], Hai YANG[4], Wei RUAN[†‡5]

*[1]College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China*

*[2]Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60208, USA*

*[3]College of Computer Science and Technology, Zhejiang University of Technology, Hangzhou 310023, China*

*[4]Magic Shield Co., Ltd., Hangzhou 310027, China*

*[5]College of Control Science and Engineering, Zhejiang University, Hangzhou 310027, China*

[†]E-mail: chunlinxiong94@zju.edu.cn; ruanwei@zju.edu.cn

**Abstract:** In recent years, PowerShell has increasingly been reported as appearing in a variety of cyber attacks. However, because the PowerShell language is dynamic by design and can construct script fragments at different levels, state-of-the-art static analysis based PowerShell attack detection approaches are inherently vulnerable to obfuscations. In this paper, we design the first generic, effective, and lightweight deobfuscation approach for PowerShell scripts. To precisely identify the obfuscated script fragments, we define obfuscation based on the differences in the impacts on the abstract syntax trees of PowerShell scripts and propose a novel emulation-based recovery technology. Furthermore, we design the first semantic-aware PowerShell attack detection system that leverages the classic objective-oriented association mining algorithm and newly identifies 31 semantic signatures. The experimental results on 2342 benign samples and 4141 malicious samples show that our deobfuscation method takes less than 0.5 s on average and increases the similarity between the obfuscated and original scripts from 0.5% to 93.2%. By deploying our deobfuscation method, the attack detection rates for Windows Defender and VirusTotal increase substantially from 0.33% and 2.65% to 78.9% and 94.0%, respectively. Moreover, our detection system outperforms both existing tools with a 96.7% true positive rate and a 0% false positive rate on average.

**Key words:** PowerShell; Abstract syntax tree; Obfuscation and deobfuscation; Malicious script detection

## 1 Introduction

Traditional malware-based attacks that target personal computers (PCs) and servers can be detected and prevented by antivirus software and strict policies because new unknown files constantly appear on the hard disk. As a result, attackers aim to avoid creating these suspicious files in the course of the attacks, and new techniques have been adopted, such as the "living off the land" and "fileless" attack techniques (Wueest and Anand, 2017). According to the white paper of CrowdStrike (2018), eight out of 10 attack vectors that resulted in a successful breach used fileless attack techniques. Among these techniques, PowerShell (Samratashok, 2020) is the most brilliant. PowerShell has been widely adopted by attackers because it (1) is a built-in Windows framework, (2) has full access to operating system

resources, and (3) is trusted by default.

As early as 2016, Symantec discovered the popularity of PowerShell among attackers (Wueest and Stephen, 2016), and PowerShell was widely adopted by attackers in subsequent attacks (Symantec, 2018), including financial threats, phishing mails, ransomware, crypto-mining, and advanced persistent threat (APT) attacks.

In response to this threat, traditional detection methods were applied, such as handpicked static string-level signatures (Diggs, 2017) and machine learning (Curtsinger et al., 2011; Hendler et al., 2018; Rusak et al., 2018). However, similar to JavaScript, PowerShell can construct and execute commands dynamically, which makes existing approaches inherently vulnerable to obfuscation (Mateas and Montfort, 2005; Barak et al., 2012). As shown in our experiment in Section 2, automatic obfuscation tools can help malicious code bypass almost all antivirus engines in VirusTotal (Google, 2004). To overcome this challenge, it is highly crucial and necessary to have an effective and lightweight deobfuscation algorithm for PowerShell that can benefit detection systems that target PowerShell attacks and can also be easily extended to other script languages.

Existing deobfuscation methods have deficiencies in many aspects; for example, they are not lightweight, and they lack generality and accuracy (Lu and Debray, 2012; Xu et al., 2012; AbdelKhalek and Shosha, 2017; Liu et al., 2018; R3MRUM, 2018; Li et al., 2019; Rubin et al., 2019; Ugarte et al., 2019). To address this critical research gap, we design and implement the first generic, effective, and lightweight deobfuscation approach for PowerShell scripts. The key insight of our approach is that the obfuscated script fragments must be recovered at runtime before they are executed. Thus, for an obfuscated script, we can obtain the original script by combining the obfuscated fragments and their corresponding recover logic and emulating the whole process. Based on this insight, we introduce a novel deobfuscation method that is based on an abstract syntax tree (AST) (FOLDOC, 1994).

To evaluate the effectiveness, generality, and efficiency of our approach, we first collected 4141 malicious samples and 2342 benign scripts and then applied our approach to this data set. The results show that the similarity of the deobfuscated scripts to the original scripts is approximately 93.2% on av- erage. Specifically, our approach can perfectly de- obfuscate scripts with script-block-level obfuscation, with a similarity of 100%. The similarity is much higher than that obtained by state-of-the-art works, including our previous work (Li et al., 2019), which shows the effectiveness of our approach. Additionally, our deobfuscation system can recover a script in less than 0.5 s on average, and it is more efficient than that in our previous work.

The results further show that the detection rates of other detection systems are significantly improved with the help of our deobfuscation approach, as shown in Section 5.2.2. Our deobfuscation method does not improve the false positive rate (0%) of Windows Defender or VirusTotal. Furthermore, we apply the objective-oriented association (OOA) detection algorithm proposed in our previous work to verify the effectiveness of this deobfuscation method.

We summarize the contributions of this paper as follows:

1. We propose the first deobfuscation method based on ASTs (Li et al., 2019). Based on this, we further refine the obfuscated script detection method and the whole deobfuscation process to make them more effective, efficient, and generic. The deobfuscation method is mainly a static technique with only emulation execution. We demonstrated its usability in Li et al. (2019) and further study the essence of obfuscation based on its impact on ASTs (Section 2.3), propose a depth-first method to traverse an obfuscated script in a top-down order to significantly improve the efficiency of deobfuscation (Section 4.2), and implement multilayer deobfuscation and deobfuscation with variables and constants (Section 4.4).

2. We are the first to propose an inspection of obfuscation techniques based on their impact on ASTs. We can then detect obfuscated scripts directly by traversing the ASTs, which does not depend on the training data set and thus makes our approach more generic.

3. We implement a deobfuscation and detection system based on our design. The system is then evaluated on a set of 6483 PowerShell samples. The results show that compared to the conference version (Li et al., 2019), our deobfuscation system is not only more effective—increasing the average similarity from approximately 80% to 93.2% (the original similarity is 0.5%, and 100% similarity is achieved with script-block deobfuscation), but

also more efficient—increasing the speed of deobfuscation by nearly 10 times. Additionally, our system is more generic and can deal with multilayer obfuscation and obfuscation with constants and variables. Furthermore, when our deobfuscation method is applied, our semantic-aware attack detection system outperforms both Windows Defender and VirusTotal with a 96.7% true positive rate and a 0% false positive rate on average. Moreover, our system will be open source.

## 2 Background and motivation

PowerShell is widely used in different tactics in real-world attacks due to its unique features (MITRE, 2015), especially in downloading and executing payloads, establishing reverse shells, and collecting victims' information. In this section, we will introduce the challenges of detecting malicious PowerShell scripts in real-world scenarios.

### 2.1 "Living off the land" and fileless attacks via PowerShell

"Living off the land" is a clear trend in targeted cyber attacks. Attackers are increasingly making use of the tools already installed on targeted computers or are running simple scripts and shellcode directly in memory. Creating fewer new files on the hard disk means that there is less chance of being detected by traditional security tools and therefore the risk of an attack being blocked is minimized (Wueest and Anand, 2017). Fileless attacks include memory-only threats (e.g., SQL Slammer), fileless persistence (e.g., Microsoft Visual Basic Script in the registry), dual-use tools (e.g., psExec.exe), and non-portable executable file attacks (e.g., Microsoft Office documents with macros or scripts).

PowerShell is one of the best choices among programming languages and scripting languages due to its unique features, as discussed in Section 1. Moreover, there are existing PowerShell exploitation frameworks (EmpireProject, 2015; Bohannon, 2017a) and it is easy for attackers to use them and bypass existing detection systems.

### 2.2 PowerShell obfuscation techniques

Obfuscation is the deliberate act of creating source or machine code that is difficult for both hu-

mans and detection systems to understand, and it is the most popular way to evade detection. In PowerShell, malicious scripts try to avoid detection by hiding their malicious intent. To achieve this, attackers take advantage of the dynamic features of PowerShell to create highly obfuscated scripts. Specifically, PowerShell scripts can be reconstructed at runtime, as shown in Fig. 1. As a result, there are as many string-manipulating methods as obfuscation methods. Logically, the process of executing obfuscated scripts can be divided into two parts: (1) Calculating strings. The output strings can be one of the multiple elements in the original scripts, such as commands and parameters. (2) Reconstructing the original scripts and executing them. When deobfuscating at the token level, these two steps are mixed up, which makes the deobfuscation more challenging. We analyze the commonly used obfuscation techniques described in Symantec's white paper (Candid, 2016), summarize them in Table 1, and discuss them according to the following three categories:

O1. Randomization. Randomized obfuscation is a set of techniques that can be used to make random changes to PowerShell scripts without affecting the original semantics and functions; they include white space randomization, case randomization, variable and function name randomization, and

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

```
# 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)**

```
# 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)**

Fig. 1 Examples of obfuscated scripts at different levels: (a) original script; (b) encoding-based obfuscation at the script block level; (c) multiple obfuscation methods at token level

Table 1  Some common obfuscation techniques and their effects

| Category | Technique | Original | Obfuscated |
|---|---|---|---|
| Randomization | White spaces | 'Task'+'list' | 'Task'+'list' |
| | Case randomization | Tasklist | TasKLiST |
| | Ignored characters | Tasklist | ta'skl'i'st |
| | Variables and function names | $count=1 | $a3T2S=1 |
| | Aliases | Clear-History | clhy |
| String manipulation | Splitting | 'Tasklist' | 'Task'+'list' |
| | Reordering | Tasklist | IEX(\|\|{2}{0}{1}" -f 'skl','ist','ta') |
| | Replacing | Tasklist | IEX ('teet').replace('ee','asklis') |
| | Variables | Tasklist | $a=asklis; IEX('t'+$a+'t') |
| | Reversing | Tasklist | IEX([string]::join(\|\|,([regex]::matches ("NoiSserPXe-eKoVni\|)'ts'+'ilks' +'at'( " ,'.' ,'righttoleft')\|%{$_.VaLue}))) |
| Encoding | ASCII/Unicode/Hex/AES | Tasklist | (new-object io.streamreader((new-object system.io.compression.deflatestream([io.memorystream][convert]::frombase64string('k0kszs7jlc4baa=='), [io.compression.compressionmode]::decompress)), [text.encoding]::ascii) ).readtoend( )\|invoke-expression |
| | Standard/customized encoding | | |

Note that not all the obfuscation techniques are listed here and that in practice, the attacker often chooses some of these techniques to obfuscate the scripts in multiple layers

the insertion of characters that are ignored by the PowerShell interpreter. The variable "Random" in Fig. 1c is an example of this kind of obfuscation. Other methods, such as function aliases, can also be classified in this category. These techniques take advantage of the features of PowerShell to make the code difficult to understand without affecting the semantics or syntax.

O2. String manipulation. String-based operations can be used to obfuscate scripts, and they include string splitting, reversing, and reordering as well as replacing variables, which refer to the calculation of variables "StrReorder," "Strjoint," and "Url" in Fig. 1c.

O3. Encoding. Attackers can use built-in decoding functions and customized encoding modules. All encoded scripts are similar; they are unintelligible strings with few semantics. Thus, this is the most popular obfuscation technique in real-world attacks. The variable "$SecstringEncoding" in Fig. 1b shows how encoding is used in obfuscation.

In practice, these methods are not used alone. They are often combined and used multiple times to improve the obfuscation effectiveness. For example, ObfuscatedEmpire (Bohannon, 2017a) uses combinations of the above techniques to build an obfuscated variant of a well-known PowerShell attack framework, PowerShell Empire (EmpireProject,

2015). It has also been reported in recent white papers (Ahl, 2017; Ackerman et al., 2018) that many attacks use combined and multilayer obfuscation methods.

## 2.3  Analyzing the impact of obfuscation based on AST

Although we have listed some common obfuscation techniques, there is still an important open question: how to distinguish obfuscated fragments from normal fragments. Existing works (Hendler et al., 2018; Li et al., 2019) use machine learning techniques or patterns to find obfuscated pieces, but these approaches depend heavily on prior knowledge and training data, which makes them non-generic, especially when facing new obfuscation techniques.

In this subsection, we give a novel characterization of obfuscation based on its impact on an AST. Obfuscation is commonly used to evade detection by hiding the original scripts that represent the attackers' intentions; the common techniques are discussed in Section 2.2. When deobfuscating, the scripts may still contain obfuscated fragments after a round of deobfuscation. Alternatively, the scripts can be overly deobfuscated, which means that the original scripts are replaced with their execution results. Therefore, we need a clear definition of obfuscation to find

obfuscated fragments and decide when to stop the deobfuscation process.

To do this, we first illustrate the obfuscation process at the AST level. As shown in Fig. 2, all obfuscation techniques can be classified into three categories based on their impact on ASTs, including enhancing, packing, and splitting. The corresponding samples are presented in Table 2. Each of these obfuscation techniques can hide the original scripts by changing the contents on nodes in ASTs. In Fig. 2, the top AST nodes represent the script blocks, including ScriptBlockAst, NamedBlockAst, and IfStatementAst, while the child nodes are single commands, parameters, attributes, etc. The analyzers and detection systems look at this tree from the top and can analyze only its top layers. If the script is not obfuscated, the semantics are obvious. Therefore, the obfuscation process hides the true scripts in the top layers with various techniques. As a result, the scripts are manipulated, and the subtrees become more complex, with new string-related AST subtrees (CommandExpressionAsts) and variables (VariableExpressionAsts). One exception here is that an assignment command contains one left-value and one right-value, which should be regarded as separate commands, and thus assignment commands cannot be top single commands. Note that although the original node is modified, the

original script should always be restored to this node at runtime to ensure the correctness of the execution.

According to the different impacts on ASTs, there are three main methods of implementing obfuscation: enhancing, packing, and splitting. Enhancing obfuscation divides a complete command into multiple fragments and reconstructs them at runtime. Packing obfuscation packs multiple commands together and recovers them at runtime. Splitting obfuscation is similar to enhancing obfuscation, but with additional variables and constants. Although there are multiple obfuscation methods, the essence of obfuscation is the same. Thus, the purpose of deobfuscation here is to reveal the hidden script, and the key to finding obfuscated scripts is to find expression ASTs (string-manipulation and variables) below the top statement nodes (CommandAsts, ParameterAsts, etc.). Note that obfuscation techniques O2 and O3 can be used to modify ASTs in all three ways.

In conclusion, expressions and variables are used to change the structures of ASTs in top layers, to hide the true semantics of the original scripts. Thus, it can be the signature to find obfuscated nodes in ASTs.

**Table 2  Three kinds of obfuscations based on the impacts on ASTs, corresponding to Fig. 2**

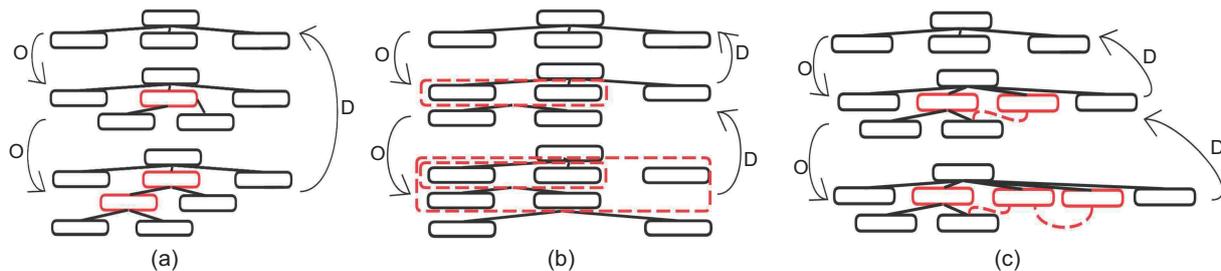| Category | Original | One-layer obfuscation | Two-layer obfuscation |
|---|---|---|---|
| Enhancing | ('tasklist') | ('{0}{1}' -f 'task','list') | ('{0}{1}' -f ('{0}{1}' -f 'ta', 'sk'), 'list') |
| Packing | tasklist<br>whoami<br>ipconfig | ('tasklist<br>whoami') \| iex<br>ipconfig | ("('tasklist<br>whoami') \| iex<br>ipconfig") \| iex |
| Splitting | Tasklist | $a='task'<br><br>($a+'list') | $b='ta'<br>$a=$b+'sk'<br>($a+'list') |



**Fig. 2  Three kinds of obfuscations based on the impacts on ASTs: (a) enhancing obfuscation; (b) packing obfuscation; (c) splitting obfuscation (O: obfuscation; D: deobfuscation)**

## 2.4 Current effectiveness of obfuscation on PowerShell attack detection

In this subsection, we experimentally evaluate the effectiveness of the state-of-the-art PowerShell attack detection systems for some common obfuscation schemes.

### 2.4.1 Experimental methodology

Because there is no existing large-scale obfuscated PowerShell data set, we chose to obfuscate the scripts ourselves. Thus, in this experiment, we manually chose four representative obfuscation schemes that combine different obfuscation techniques and obfuscate the scripts at different levels, the token level and the script-block level. Token-level obfuscations try to manipulate each token in the script separately, whereas script-block-level obfuscations treat the script as a whole and try to manipulate the whole script. The details of the obfuscation schemes are given in Table 3. Specifically, we adopted two encoding techniques, secure string-based encoding (Secstring) and hex-based encoding (Hex), both of which are commonly used in real-world attacks (Wueest and Stephen, 2016). We used a famous open-source PowerShell obfuscation framework, Invoke-Obfuscation (Bohannon, 2017b), which is widely used in APT attacks (Ahl, 2017; Ackerman et al., 2018), to implement these schemes.

For the PowerShell samples, we collected 75 un-obfuscated malicious scripts and the same number of benign samples from repositories on GitHub and security blogs and then obfuscated each of them using the four schemes above. We then uploaded all the original scripts and the obfuscated scripts to Virus-Total, a website that aggregates 70+ well-known antivirus products and online scan engines for online virus checking. Finally, we counted the number of products and engines that reported alerts to evaluate their effectiveness.

**Table 3   Obfuscation schemes**

| Scheme | Adopted obfuscation techniques (Section 2.2) |
| --- | --- |
| S1 | O1, O2 (token level) |
| S2 | O1, O2 (script-block level) |
| S3 | O1, O3 (script-block level, string encoding) |
| S4 | O1, O3 (script-block level, hex encoding) |

### 2.4.2 Results

As shown in Fig. 3, all four obfuscation schemes can reduce the detection rate significantly. Specifically, the average number of alerts dropped significantly, from 13.2 to at most 3.1, especially for S1 (string manipulation at the token level). Although the alert numbers of malicious scripts for S3 and S4 were higher than those for S1 and S2, this does not show that some engines have the ability to detect obfuscated malicious scripts. Instead, they merely detect and report obfuscation patterns; benign scripts that use S3 and S4 are also reported by them. Because obfuscation has benign uses, such as protecting intellectual property and avoiding unwanted changes (Tobias, 2018), detecting obfuscation features cannot be equivalent to detecting malicious scripts. This would lead to false alerts.
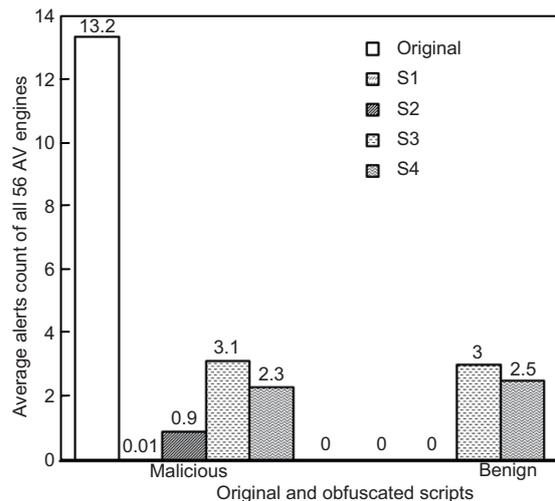


**Fig. 3   Average alerts count on VirusTotal for the original scripts compared with the obfuscated scripts (Only 56 out of the 70 AV engines returned detection results of the samples)**

## 3 Overview

As shown in Section 2.4, none of the state-of-the-art PowerShell attack detection methods is free of the influence of obfuscation techniques. Thus, there is an urgent need to design and implement a generic, effective, and lightweight deobfuscation method for PowerShell.

In this study, we design such a deobfuscation method, and then develop a semantic-aware PowerShell attack detection system based on it. As shown

in Fig. 4, the whole process can be divided into three phases:

Deobfuscation phase. In this phase, we propose a novel approach based on AST and emulation execution. The whole deobfuscation phase is based on AST, including obfuscated fragment detection, emulation-based deobfuscation, and final script reconstruction. The deobfuscation method can improve not only the detection method proposed in our previous paper (Li et al., 2019), but also existing detection, analysis, and forensics systems for PowerShell. This method can also be extended to support other scripting languages, such as JavaScript and Python, which we believe constitutes a general contribution to the research area of scripting languages.

Training and detection phases. Scripting languages are designed to be simple; thus, it is much easier to extract their semantics according to their commands than it is for other programming languages (Kannumittal, 2018). After deobfuscation, the semantics of the scripts are exposed; thus, we design and implement the first semantic-aware malicious PowerShell detection system. We extract 31 new and unique OOA rules as malicious signatures by adopting classic OOA mining (Shen et al., 2002). The process is shown in Fig. 4.
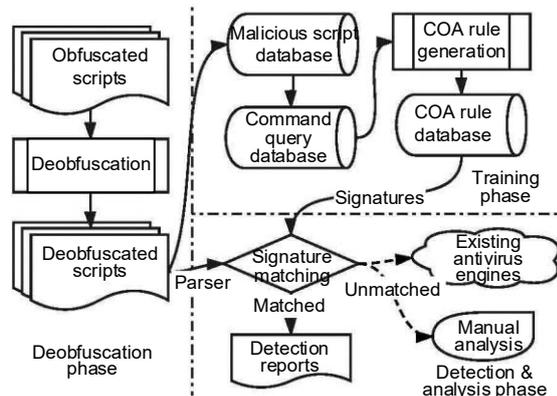


**Fig. 4  The framework of our deobfuscation approach and semantic-aware PowerShell attack detection**

Our AST-based deobfuscation approach is a combination of static and dynamic analysis. Each phase is static except for emulation-based execution, which is used to obtain the string-type return values, and these strings may be the original scripts. Compared to dynamic approaches (R3MRUM, 2018; Rubin et al., 2019), our approach is more efficient, requires much less overhead, achieves higher code

coverage and greater stability, and does not need any modification of the PowerShell interpreter or the operating system. Compared to static approaches (Curtsinger et al., 2011; Diggs, 2017; Hendler et al., 2018; Rusak et al., 2018; Ugarte et al., 2019), our approach provides more insight into obfuscation, which makes it more resilient to different obfuscation techniques. Also, based on our semantic-aware detection approach, the detection results are more explainable. With all these advantages, this method can be used in different application scenarios, especially in the following:

Real-time attack detection. Benefiting from the accuracy and efficiency of this method, our deobfuscation and detection systems can be deployed to deobfuscate, detect, and analyze PowerShell scripts in real time. For example, we have developed a real-time detection system based on event tracing for Windows (ETW), which can provide Power-Shell scripts that are executed from the "Microsoft-Windows-PowerShell" provider.

Large-scale automated script analysis. Because there are no other mature deobfuscation methods, the existing automated PowerShell analysis platforms (Crowdstrike, 2014) depend on manual static string based signatures or dynamic analysis. The former method is limited by prior knowledge and can be vulnerable to obfuscations, whereas the latter cannot guarantee code coverage and is vulnerable to dynamic evasion techniques (such as delayed execution and logic bombs). With our approach, the analysis can be resilient to obfuscation and include more detailed semantics information.

## 4  PowerShell deobfuscation

In this section, we first describe the key idea behind our approach, and then describe the design details. Existing works are either based on manual analysis of specific obfuscation techniques (Liu et al., 2018) or can work only on a subset of obfuscation schemes (Bohannon, 2017b; R3MRUM, 2018). By combining dynamic and static techniques, our approach addresses most of the existing obfuscation techniques using dynamic techniques and is highly efficient in recovering the original scripts using the static advantage of static techniques. Furthermore, although this approach was developed for Power-Shell, it can be easily extended to other scripting

languages (e.g., JavaScript).

One key challenge of PowerShell deobfuscation is recovering dynamically generated script fragments. For example, in Fig. 1b, attackers encode an original script fragment in an SHA (secure hash algorithm) hash value and recover it by decoding the hash value dynamically. Pure static methods cannot deal with these dynamic codes, because the original scripts cannot be recovered without being executed. To address this challenge, we propose an emulation-based recovery method. The key idea behind our approach is that the obfuscated script fragments must be recovered as the original, unobfuscated scripts before being executed by the PowerShell interpreter during runtime. As shown in Fig. 1, obfuscated scripts can always be divided into two parts: obfuscated fragments and corresponding recovery processes. As long as these obfuscated fragments are found, we can use the PowerShell interpreter to emulate the recovery process dynamically.

Another key challenge is locating obfuscated fragments in practice because of the extremely complex code structure, especially when the script is obfuscated in multiple layers. To address this challenge, we propose a novel AST-based obfuscation-locating method to efficiently find these recoverable fragments by traversing the AST nodes with static program analysis techniques and then deobfuscating them by reconstructing the AST. Specifically, AST is a tree representation of the abstract syntactic structure of source code written in a programming language. With the help of ASTs, the structure and syntax of scripts are much easier to identify.

## 4.1 Deobfuscation overview

The overall workflow of our deobfuscation system is shown in Fig. 5, and has four major modules: AST-based obfuscation-locating module, emulation-based recovery module, updating module, and reconstruction module. Specifically, the AST-based obfuscation-locating module (Section 4.2) locates obfuscated fragments in a given script by parsing the script to an AST and traversing the AST to record contextual information. Then the located obfuscated fragments are sent to the emulation-based recovery module (Section 4.3) to be recovered as deobfuscated PowerShell fragments. After recovery, the updating module (Section 4.4) replaces the obfuscated fragments in AST with the deobfuscated fragments. Note that the original scripts might be obfuscated in multiple layers, which means that the deobfuscated fragments in the last step may still contain embedded obfuscated fragments. To address this challenge, our system applies the above three steps against the updated AST until no obfuscated fragments remain. Finally, we obtain an AST that does not contain any obfuscated fragments and the reconstruction module (Section 4.5) reconstructs the original PowerShell scripts from AST.

We reuse the obfuscated script in Fig. 1b as an example to introduce our deobfuscation method step by step, as shown in Fig. 6. We describe the figure
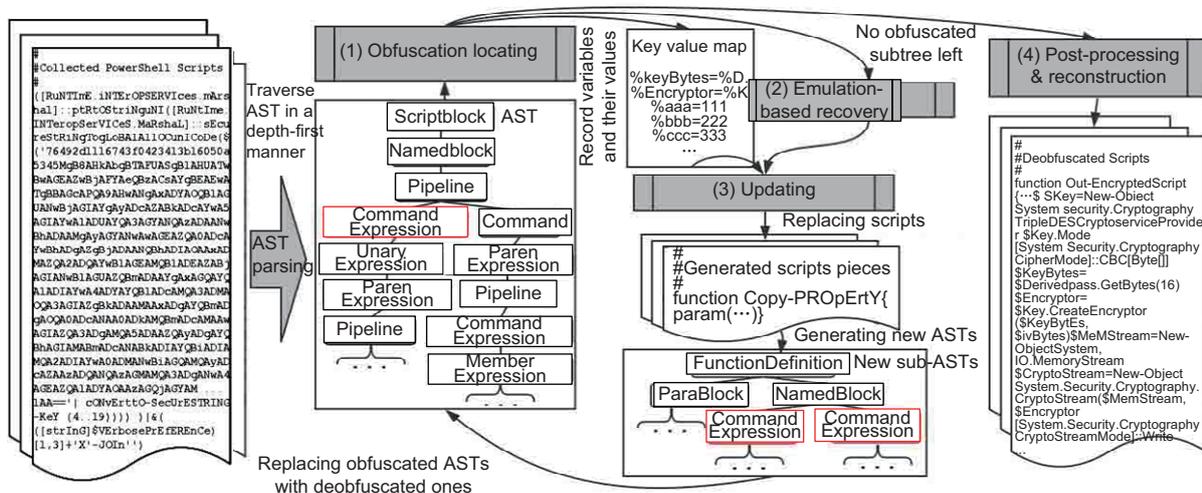


Fig. 5 An overview of the proposed subtree-based deobfuscation for PowerShell scripts

in detail in the following subsections. First, the obfuscated script is parsed into a corresponding AST, as shown in Fig. 6a. The rectangle nodes (③⑥) correspond to complete statements, and the diamond node (②) is an expression. The oval nodes (①⑤) represent variables and the remaining one (④) is a constant string, which can be data or a function name. Second, obfuscated fragments are located and marked by traversing the AST; for example, the expression node ② under a complete command node ③ is regarded as an obfuscated node. Third, obfuscated fragment ② is sent to an isolated PowerShell session, and it is replaced by the return value, as shown in Fig. 6b node ⑦. Fourth, when traversing the updated AST in the second round, the variable ① and its value are recorded, because the value is a constant string after the last step. As a result, node ⑤ is replaced by its value (Fig. 6c, node ⑧). In this AST, we find that the constant string ⑧ is under a pipeline node ⑥ that has a left child node ④, which represents the dynamic execution function "Invoke-Expression." Thus, the script in node ⑧ would be executed dynamically in runtime. However, whatever the script is, it can be regarded only as a constant string by the PowerShell interpreter, and the script can still be obfuscated. Therefore, this dynamic command ⑥ is converted to its original form by the updating module (Fig. 6d). Finally,



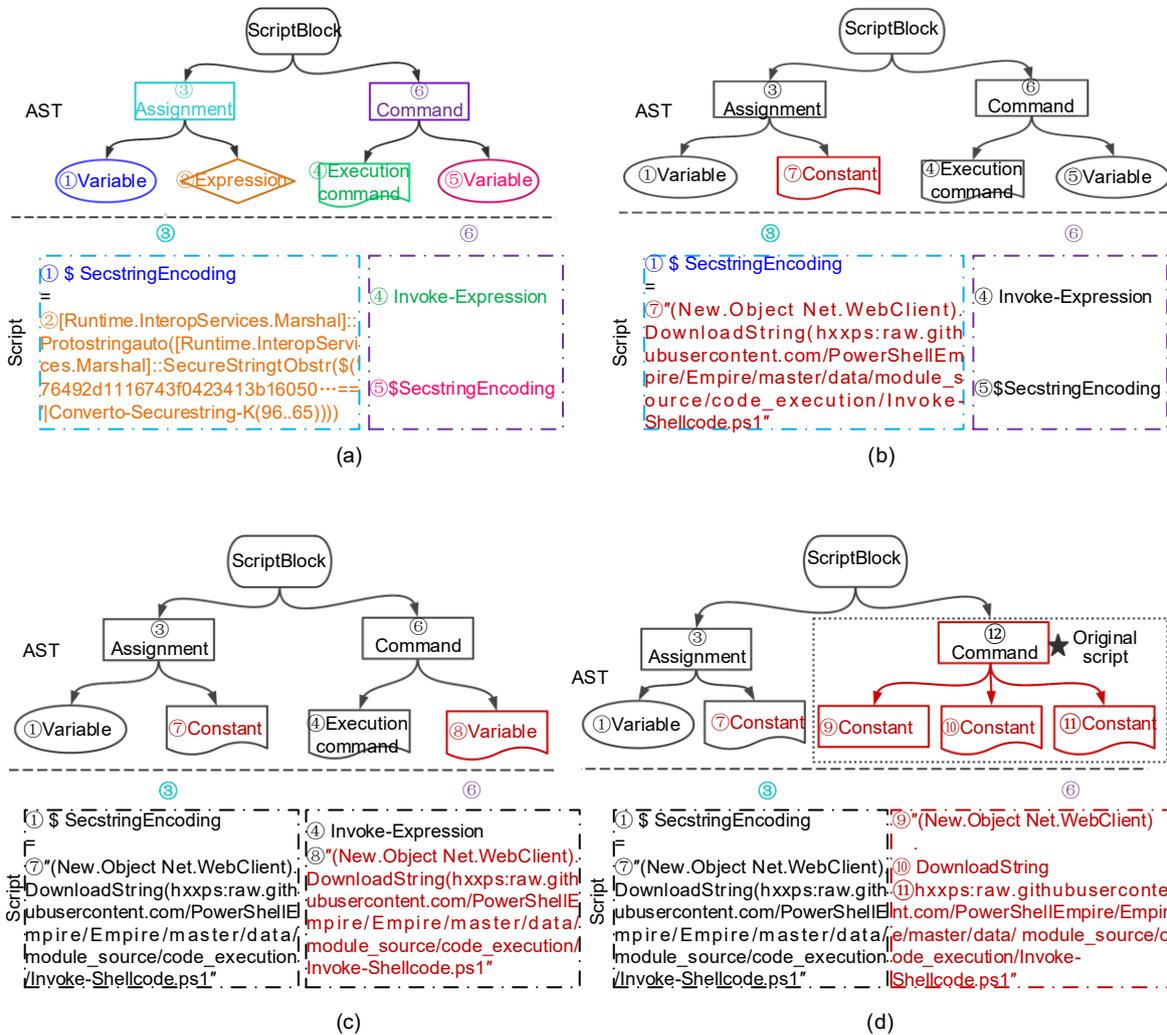**Fig. 6 The simplified ASTs during the deobfuscation of the sample in Fig. 1b: (a) obfuscated AST in Fig 1b; (b) deobfuscated AST after emulation-based recovery and replacement, in the 1st round; (c) deobfuscated AST after variable recording and replacement, in the 2nd round; (d) deobfuscated AST after updating, in the 2nd round. References to color refer to the online version of this figure**

because no obfuscated fragments are left, AST is reconstructed as the deobfuscated script. The details are described in the following.

## 4.2 AST-based obfuscation locating

In this subsection, we describe the method of locating obfuscated fragments in a given script by parsing the script into an AST and traversing the AST to record contextual information.

### 4.2.1 AST parsing

Locating obfuscated fragments in plain text PowerShell scripts is difficult. However, a structured format can be used to better analyze the scripts. Thus, we first convert plain text scripts into structured format. Specifically, we leverage an official library System.Management.Automation.Language to parse an input PowerShell script into an AST. An AST is a tree that represents the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a grammatical element that occurs in the source code, and the type of each node is represented in the AST. Such structured information is useful for code analysis.

As shown in Fig. 6a, the obfuscated script in Fig. 1b is parsed into an AST. The lower part of Fig. 6a is the PowerShell script and the upper part is the corresponding AST. Each element in the script is parsed into an AST node and we use the same color to represent this correlation relationship. The script first decodes an SHA hash value into a string and then executes the string as a command. The whole script contains two statements: an assignment statement ③ and a command ⑥ statement. The assignment statement stores the return value of expression ② in a variable ①. The command statement inputs the value stored in variable ⑤ to the execution command ④ to execute it as a command.

### 4.2.2 Variable recording

Variables and constants can also be introduced during obfuscation to further hide the original semantics. As shown in Fig. 1c, the original script is divided into four variables and it is reconstructed in runtime to be executed. In addition, the original script cannot be recovered unless the variable node ⑤ in Fig. 6 is replaced by its value.

We record all this contextual information while traversing the AST. However, this kind of obfuscation can be very complicated: loops, branches, nesting, and assignments among several variables make deobfuscation difficult. In practice, although it is possible to cover all possibilities in a static analysis, we record only variables with constant string values for efficiency, and these can include all the samples generated by Invoke-Obfuscation (Bohannon, 2016). The shortcoming of this simplification is that it cannot recover the values of variables that involve control flows with branches.

### 4.2.3 Obfuscation locating

As we discussed in Section 2.3, it is difficult to distinguish obfuscated fragments in scripts. However, with the impact of obfuscation analysis based on the AST, we summarize the AST-based obfuscation-locating method here: the obfuscated nodes are these "expression nodes" under "statement nodes."

In PowerShell, "expression nodes" are used to hide semantics in scripts by introducing expressions and variables, such as "CommandExpressionAst," "BinaryExpressionAst," and "VariableExpressionAst." "Statement nodes" represent complete statements in the original scripts, such as commands, parameters, and pipelines (CommandAsts, ParameterAsts, and PipelineAsts). For example, as shown in Fig. 6b, the expression node ② under assignment command node ③ is an obfuscated node.

A typical script, thousands of bytes in size, can have thousands of nodes in AST, which means that there are thousands of subtrees, so it is very time-consuming to check all these subtrees. Fortunately, only command expressions and variables can be used to complicate these trees, which we call suspicious subtrees. Therefore, we need only to check whether there are suspicious subtrees below a top statement node. Leveraging this insight, the number of subtrees we need to check is significantly reduced. Based on this idea, we deobfuscate the suspicious subtrees while traversing the whole AST in a depth-first manner. Another benefit of depth-first traversal is that it is much faster than the original bottom-up traversal method because once we find an obfuscated node in the upper layer, we can obtain the original script in one step instead of deobfuscating the scripts layer by layer.

## 4.3  Emulation-based recovery

In this subsection, we describe how the emulation-based method recovers obfuscated fragments located in the last section to the original PowerShell scripts. Original scripts are hidden by various string manipulation and encoding methods in an obfuscation process (e.g., an original PowerShell script in Fig. 1a is obfuscated in an SHA value "76492d1116743f0423413b16050...==" in Fig. 6a script ②). It is difficult to enumerate each obfuscation technique and develop a corresponding method to deobfuscate the script. However, these obfuscated fragments must be recovered to the original scripts at runtime so that the fragments can be executed by the PowerShell interpreter (e.g., the attack in Fig. 6a leverages the built-in SecureStringToBSTR API to recover the SHA value to the original fragments). Based on this observation, we propose an emulation-based recovery method using the native PowerShell interpreter to emulate the recovery process and obtain the recovered original fragments. Specifically, we set up a PowerShell execution session and execute the obfuscated fragments that were detected in the last step (e.g., decoding the SHA value in an isolated PowerShell session). The return value of the execution is a string that corresponds to the recovered PowerShell fragment.

## 4.4  Updating

In this step, the obfuscated fragments in the AST are replaced by the original PowerShell fragments recovered by the emulation. When there are AST nodes of variables and constants under a statement node as right child nodes, we can replace the variables with their values. For example, node ⑤ in Fig. 6a is replaced, while node ① is not.

As is generally known, scripts can be divided into two parts: commands and data. In this problem, if the obfuscated fragments are data in the original scripts, they can be replaced with the recovered strings directly during deobfuscation. However, if they are commands, we should first find the corresponding original nodes that cause these recovered strings to be executed. In this stage, the deobfuscated scripts are recovered as strings (e.g., the value of "$SecstringEncoding" in Fig. 6a can be obtained by emulation-based recovery and replacement, and

it is a string corresponding to the original script in Fig. 1a). There is another command to execute the string as commands dynamically (e.g., Fig. 6a node ④). However, the PowerShell interpreter can identify only the dynamic commands as a string in this stage as shown in Fig. 6c node ⑧, and thus cannot analyze the syntax of dynamic commands, which hinders further analysis, especially for multilayer deobfuscation. Thus, we must first convert the dynamic commands to their original form in ASTs (Fig. 6d).

Fortunately, there are only three ways to execute a string as a command in PowerShell ('.', '&', and 'invoke-expression'). In this stage, we use regex patterns to convert strings to commands. Then, because new commands are uncovered in AST, the recovered commands should be parsed into new subtrees and be used to replace the obfuscated nodes. For example, "command" ⑥ in Fig. 6c consists of a dynamic execution command ④ and a constant string ⑧. It is parsed and replaced by a new AST subtree in Fig. 6d. New subtrees may still have obfuscated fragments, so they should be deobfuscated until no obfuscated fragments are left.

## 4.5  Reconstruction

After the last process, there are no obfuscated fragments in AST, and we obtain a script that has the same semantics as the original one. However, in terms of syntax, there are still differences between these two scripts. These differences are introduced mainly by the obfuscation process. For example, extra parentheses and braces are added during the process. In the post-processing step, the syntax-level changes introduced by the obfuscation process are located with regular expressions and fixed accordingly. For each AST, there is only one corresponding script that can also be parsed into that AST. Therefore, after the post-processing step, the deobfuscated AST can be reconstructed as its corresponding script, which is the final output of our deobfuscation process.

## 4.6  Comparison to our previous work

We have improved our previous approach in several aspects: effectiveness, efficiency, and universality. We summarize the reasons and the corresponding design improvements in this work as follows:

First, it is not proper to pick obfuscated scripts

in PipelineAst nodes during emulation-based recovery. When executing scripts in this kind of AST, some of the original commands would be executed directly by mistake. For example, the command "Invoke-Expression ('{0}{1}{2}' -f 'stop', '-', 'Computer')" is used to shut down the computer by calling "Stop-Computer" dynamically, and it would be recognized as a PipelineAst. When trying to execute this command during emulation-based recovery in deobfuscation, the computer will be terminated. Therefore, this script cannot be deobfuscated by the method in our previous work. Even worse, the deobfuscation process would lead to disaster in real-world scenarios. In this work, our design deobfuscates in CommandExpressionAsts, which can solve this problem. The true command "Stop-Computer" would be recovered by our system.

Second, using novel obfuscation inspection based on their impact on ASTs, we can detect obfuscated scripts directly by traversing the ASTs, which is more generic and efficient, instead of by adopting machine learning modules, which depends heavily on manually labeled training data and takes more time in finding obfuscated pieces.

Third, when we first came to this problem, we believed that it is a reasonable idea to deobfuscate scripts layer by layer; in other words, scripts are deobfuscated from the bottom up, as shown in Fig. 2. This is what we did in our previous work (Li et al., 2019). However, this approach makes the deobfuscation process traverse all the nodes in the ASTs to make sure that the root node, which corresponds to

the whole script, is totally deobfuscated in the end. In this study, with the help of novel inspection techniques, we propose a top-down deobfuscation process that makes the current process hundreds of times faster than that in our previous work in some complicated cases (Fig. 7).

Fourth, some deobfuscated scripts have few similarities when compared to the original scripts, mainly because of obfuscations involving variables and inaccurate deobfuscation processes. We refine the design in this study, which improves the similarity results.

Finally, in this study, we solve the multilayer deobfuscation problem which cannot be addressed by our previous work, as discussed in Section 5.2.4.

## 4.7 Semantic-aware PowerShell attack detection

To effectively identify malicious PowerShell scripts, we propose the first semantic-aware PowerShell attack detection method based on OOA data mining in the conference version of this work (Li et al., 2019).

For programming languages, such as C++ and Java, researchers usually use several kinds of graphs, such as dependency graphs (Fredrikson et al., 2010) and control-flow graphs (Christodorescu et al., 2005), instead of API sets, to represent semantics. This is because APIs in programming languages contain only low-level semantics and can thus be ambiguous. In contrast, compared to high-level programming languages, scripting languages are
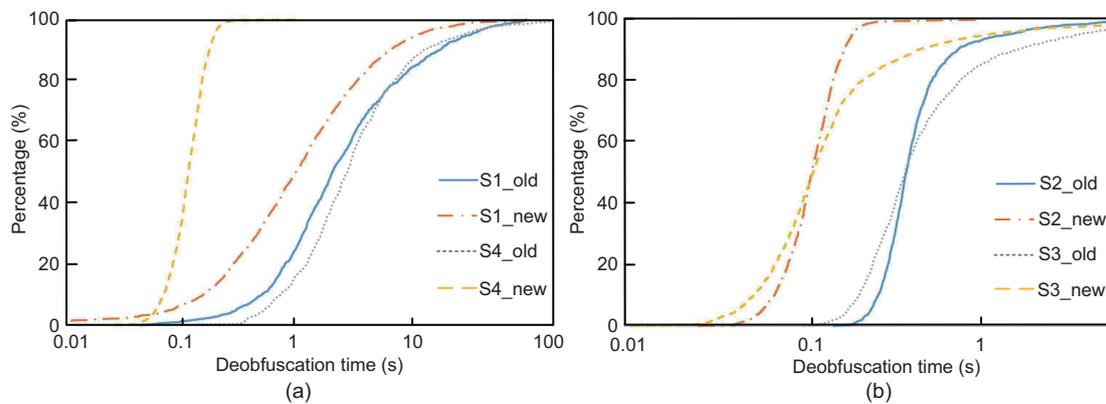


**Fig. 7 Comparison of deobfuscation efficiency in different schemes: (a) S1 and S4; (b) S2 and S3. The data is represented by the cumulative distribution function (CDF). We limit the maximum time because in some complex cases, the process takes minutes. Results show that the new method has better performance than the old one in all schemes**

designed to be much simpler (Kannumittal, 2018), such as PowerShell and JavaScript. The script is just like a set of commands, so it is easy to find the functionality of a given unobfuscated script by checking the functions called by the script.

As shown in Fig. 8, our detection system can be divided into two phases: training and detection. In the training phase, we normalize the malicious scripts by (1) converting to lowercase, (2) deleting meaningless information, and (3) checking alias. We employ a frequent pattern (FP) growth algorithm based on a frequent pattern tree (Borgelt, 2005) to generate frequent patterns and a classic classification based on OOA mining (Ye et al., 2008) on item sets 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. The representative examples and descriptions of newly identified OOA rules are listed in Table 4. Details can be seen in Li et al. (2019).
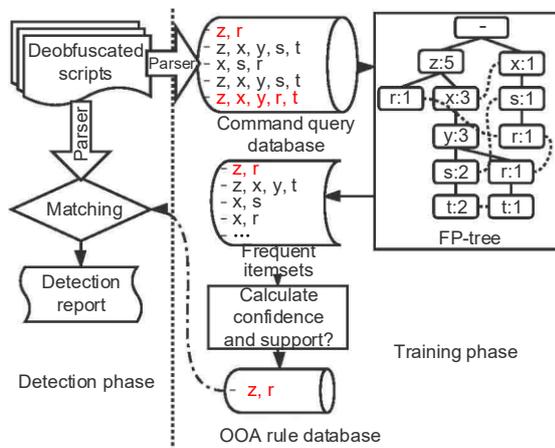


**Fig. 8 Semantic-aware detection workflow**

**Table 4 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 network string |
| DownloadFile, start-process | Download execution |
| UseshellExecute, tcpClient, redirectStandardOutput, getStream, getString, invoke-expression, ... | Reverse shell |

# 5 Experimental evaluation

## 5.1 Evaluation methodology

In this study, we evaluate the performance of our subtree-based deobfuscation method in four parts. First, we evaluate its accuracy in comparison with our previous method (Li et al., 2019), since we propose a new method of identifying obfuscated subtrees in this study. Second, we evaluate the deobfuscation quality of our system by determining the similarity between the original scripts and the deobfuscated scripts. In this step, we extend a similarity comparison algorithm proposed by Koschke et al. (2006), which was discussed in Li et al. (2019). Third, we evaluate the benefits of our deobfuscation system in comparison to other detection methods. In Section 2.2, we showed that existing malware engines are vulnerable to obfuscation. Next, we evaluate the effectiveness of our method by comparing the detection results before and after employing our deobfuscation process. In addition, we evaluate the effectiveness of the semantic-based detection algorithm proposed in Section 4.7.

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, and open-source PowerShell attack repositories, instead of intentionally making selections among them.

1. Benign samples. To collect benign PowerShell Scripts, we download the top 500 repositories on GitHub under PowerShell language type using the Chrome add-on Web Scraper (Scraper, 2019). We then find 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.

2. Malicious samples. The malicious scripts we use to evaluate detection are based on recovered scripts, and consist of two parts: (1) There are 4098 unique real-world attack samples collected from security blogs and attack analysis white papers (Diggs, 2017). These samples are limited by the method of data collection, and 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 another 43 samples from three famous open-source attack

repositories, namely, PowerSploit (PowerShellMafia, 2012), PowerShell Empire (EmpireProject, 2015), and PowerShell-RAT (Maniar, 2018).

3. Obfuscated samples. In addition to the collected real-world malicious samples, which are already obfuscated, we construct obfuscated samples by combining obfuscation methods and non-obfuscated scripts. Specifically, we deploy four obfuscation methods in invoke-obfuscation, as mentioned in Section 2.4, namely, token-based, string-based, hex-encoding, and security string-encoding on 2342 benign samples and 75 malicious samples. After this step, a total of 9668 obfuscated samples are generated. To further evaluate the ability of the methods to deobfuscate multilayer obfuscated scripts, in addition to the collected real-world malicious samples, we obfuscate all collected scripts with three layers of randomly chosen obfuscation schemes from Table 3. As a result, 4834 multilayer obfuscated samples are generated.

## 5.2 Evaluation results

In this subsection, we evaluate the effectiveness and efficiency of our approach using the collected PowerShell samples described earlier. The experimental results are obtained using a PC with an Intel Core i5-7400 processor 3.5 GHz, 4 cores, and 16 GB memory, running Windows 10 64-bit Professional.

### 5.2.1 Obfuscation detection accuracy

For obfuscation detection, we applied logistic regression with a gradient descent binary classifier based on three levels of features in our previous work (Li et al., 2019). Other machine learning techniques can also be used to find a better module that distinguishes the obfuscated AST nodes more accurately. However, with a machine learning technique, we cannot guarantee the generality of our model due to the

weak evidence and limited data set. In particular, the unknown obfuscation methods out of the training data set cannot be recognized by the machine learning module. When there are new obfuscation samples, we should retrain the whole module, which is time-consuming. Furthermore, when a sample is misclassified, it leads to incorrect deobfuscation and detection results, and it is difficult to recognize this and analyze the reason for it. Thus, as described in Section 2.3, we update the definition of obfuscation and propose a more exact way to find obfuscated fragments. In contrast to the machine learning method, we can cover all the true positive samples. We manually check the samples that are not covered by our previous method and do not regard them as false alarms, because these scripts hide their original commands and cannot be recognized directly.

As a comparison, PSDEM (Liu et al., 2018) uses a series of regular expressions combined with some syntactic information to locate obfuscated script fragments. However, this enumeration method cannot guarantee coverage.

### 5.2.2 Recovery quality

Next, we evaluate the overall recovery quality by comparing the similarity between the obfuscated sample scripts and the original ones before and after deobfuscation by employing suffix tree matching based on ASTs (Koschke et al., 2006; Li et al., 2019). Both the suffix tree and AST have been widely used in similarity calculation. The results are shown in Table 5.

As shown, after deobfuscation using our approach, the scripts obfuscated with schemes S2, S3, and S4 can be recovered perfectly, with 100% similarity scores. The average similarity increases significantly, from 0.5% to 93.2%, which is much better than the 79.7% obtained in Li et al. (2019).

**Table 5  The average similarities of obfuscated, deobfuscated, and original ASTs**

| Obf. scheme | Similiarity | | | | |
|---|---|---|---|---|---|
| | Obf. | Deobf. (new)* | Deobf. (old)** | Deobf. (PSDEM) | Deobf. (PowerDrive) |
| S1 | 1.8% | 72.8% | 71.5% | 70.6% | 1.32% |
| S2 | 0.1% | 100% | 79.0% | 79.5% | 54.2% |
| S3 | 0.01% | 100% | 82.9% | 0.01% | 84.1% |
| S4 | 0.004% | 100% | 85.2% | 0.004% | 80.7% |
| Overall | 0.5% | 93.2% | 79.7% | 37.5% | 55.1% |

Obf.: Obfuscation; Deobf.: Deobfuscation. *This paper; **Li et al. (2019)

For all deobfuscation schemes, the similarities for the scripts recovered from S2, S3, and S4 are higher than 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 can thus be deobfuscated perfectly. Note that, as indicated by the similarity scores, the deobfuscated scripts are not exactly the same as the original scripts. This is mainly because of the syntax-level changes in the obfuscation processes, e.g., extra parentheses and braces, which do not affect the semantic-aware attack detection or the understanding of the functionality, as shown later in Section 5.2.5. Additionally, some syntax-level changes are introduced during S1 obfuscation, such as additional variables and alternative methods of expressing the same command. These changes make the similarities of S1 scripts much lower than those of scripts with the three other schemes, but do not affect the semantics.

In comparison, PSDEM can cover only the first two schemes and has a lower recovery quality compared to our approach. PSDEM can do nothing for obfuscation techniques that are out of PSDEM's scope. Moreover, PSDEM does not provide an automatic method of determining the correct order in which the deobfuscation logic should be applied. Thus, for multilevel obfuscated samples, manual analysis is necessary to decide the correct order of deobfuscation logic first, whereas our approach can automatically handle such samples. PowerDrive can cover only the last two schemes and part of the samples with S2. This method depends heavily on prior knowledge and requires manually generated regex expressions to address each of the obfuscation techniques, so it cannot identify or address samples that are out of its scope.

### 5.2.3 Deobfuscation efficiency

In our previous work, we evaluated only the average time of deobfuscation. Although this time depends on the complexity of the obfuscated scripts, namely, the obfuscation schemes and obfuscated subtrees, we compare the efficiency of our system with that in the previous work. Fig. 7 shows the time required to deobfuscate one obfuscated script, and the results show that the new method performs better than the old one in all four schemes. Specifically, the new deobfuscation method reduces the time by 72.8% on average, and more than 98.3% of all the samples take less than one second to finish the deobfuscation process.

### 5.2.4 Multiple layer deobfuscation

According to the real-world malicious scripts we collected, packing obfuscation is commonly used to generate multilayer obfuscated scripts. To deobfuscate them, we should unpack the hidden scripts layer by layer. Thus, in theory, if we can deobfuscate one layer perfectly, we can deobfuscate multilayer obfuscated scripts. In practice, we generate three-layer obfuscated scripts randomly with schemes, and then deobfuscate them. The results show that we can deobfuscate these scripts with 100% similarity and the deobfuscation time grows linearly with the number of obfuscation layers.

### 5.2.5 Attack detection based on deobfuscated scripts

Table 6 shows the impact of deobfuscation on detection. Here we use the same data set as in Section 2.3. The rest of the unobfuscated samples are used as a training set for our detection system. We submit the samples separately to Microsoft Online Defender (Microsoft, 2014) and VirusTotal (Google, 2004). For VirusTotal, as long as one of the AV

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

| Sample | | Detection rate | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | WD | Deob.+WD (old) | Deob.+WD | VT | Deob.+VT (old) | Deob.+VT | Deob.+Ours (old) | Deob.+Ours (new) |
| Malicious | Original | 89.3% | 89.3% | 89.3% | 100% | 100% | 100% | 98.7% | 98.7% |
| | S1 | 0.0% | 48.0% | 48.0% | 0.0% | 76.0% | 76.0% | 90.7% | 90.7% |
| | S2 | 1.3% | 78.6% | 89.3% | 8.0% | 90.6% | 100% | 93.3% | 98.7% |
| | S3 | 0.0% | 84.0% | 89.3% | 2.6% | 96.0% | 100% | 92.0% | 98.7% |
| | S4 | 0.0% | 89.3% | 89.3% | 0.0% | 97.3% | 100% | 93.3% | 98.7% |
| Benign | Original & all schemes | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |

WD: Windows Defender; VT: VirusTotal; Deob.: scripts deobfuscated by our approach

engines detects it, we consider it to be detected. We also exclude three engines, namely, Kaspersky, ZoneAlarm, and Sophos AV, which detect obfuscation rather than maliciousness. These engines' false positives are so numerous that their results have no reference value.

As shown in Table 6, obfuscation can bypass detection effectively. For Windows Defender, the detection rate decreases by a factor of approximately 68, from 89% to 1.3%. VirusTotal performs slightly better but still fails in most cases. Its detection rate decreases by at least 12.5%. In contract, our approach is almost unaffected by obfuscation. The detection rate decreases by up to 2%. Moreover, our deobfuscation module can be integrated into existing detection systems to achieve better detection rates. For all the obfuscation schemes, deobfuscation can significantly improve the detection rate of Defender and VirusTotal. The detection rate increases by at least 48% and 76.0% for Defender and VirusTotal, respectively.

Furthermore, among the four obfuscation schemes, scheme S2 fails most often, because it is based on a string split. If scripts are not split finely enough, they can still match the signatures. Moreover, the deobfuscation effect for scheme S1 is the worst. This is because the three other schemes are script block based; the obfuscation does not change the structure of the script block, and the structure remains intact after deobfuscation. As shown in Table 6, our deobfuscation process can recover the scripts that are obfuscated with S2, S3, and S4 perfectly with 100% similarity; thus, the detection rates of the deobfuscated scripts can be as high as those of the originals ones. In addition, no detection approach has false positives, since the PowerShell scripts' structure is relatively simple and has no ambiguity.

Overall, deobfuscation can significantly improve the detection effect. Our semantic-based detection method also has good results, which means that se-

mantic analysis of deobfuscated scripts is feasible.

## 5.3 Comparison with state-of-the-art PowerShell detection approaches

Rusak et al. (2018) and Hendler et al. (2018) presented the latest detection approaches for PowerShell, applying AST- and character-based features for detection, respectively. We reproduce these two approaches to compare them with our approach. For the approach proposed by Hendler et al. (2018), the original design can support several different classifiers, and we choose only the one with the best results in their paper, i.e., the combination of a three-layer convolutional neural network and traditional three-gram. In the training of these two approaches, we use the same training set and testing set as described above. The results are shown in Table 7.

As shown, both the AST- and character-based detection approaches are bypassed by obfuscation. Our deobfuscation system can increase their true positive rate (TPR) by 87.2%. Once these scripts are deobfuscated, our results show that these two approaches can achieve similar or even higher TPRs than our approach. However, because the features used by these two approaches are at the syntax level, they can be more easily evaded than our semantic-aware approach. To show this, we simply mix benign pieces into the malicious samples at the granularity of script lines, which changes the AST structure and character distributions without affecting the script behavior. In Table 7, we call the results "mixed scripts;" these mixed scripts can greatly decrease the TPRs of the AST- and character-based detection approaches but cannot affect that of our approach.

## 6 Discussions

### 6.1 Generality of our approach

Although in this study we propose a novel AST-level emulation based deobfuscation method for

Table 7   Comparison with state-of-the-art detection approaches in terms of the true positive rate (TPR)

| Detection approach | TPR | | |
|---|---|---|---|
| | Obfuscated script | Deobfuscated script | Mixed script |
| Our approach | – | 96.7% | 96.7% |
| AST-based (Rusak et al., 2018) | 0.0% | 90.7% | 9.6% |
| Character-based (Hendler et al., 2018) | 12.1% | 95.7% | 34.7% |

PowerShell, the process of our method is independent of any special feature of PowerShell. Our method depends on the impact of deobfuscation over an AST, which is common among scripting languages. Similar obfuscation techniques are adopted for JavaScript (Kachalov, 2016). Our method needs only a parser that can convert the scripts to ASTs and an unmodified interpreter for emulation. There are existing tools for parsing JavaScript as ASTs (Google, 2011; Hidayat, 2012; Acornjs, 2013; AST Explorer, 2015; Mishoo, 2015; ShapeSecurity, 2015). With a JavaScript interpreter, our method can be transplanted to JavaScript.

## 6.2 Irreversible obfuscation

Some obfuscation processes are irreversible, such as the following:

1. Randomization: replacing variables and function names.

2. Logic structure obfuscation: (1) inserting instructions that are independent of functionality; (2) adding or changing some conditional branches.

3. Code virtualization: compiling the code as assembly/common language infrastructure (CLI), and then employing irreversible obfuscation.

The proposed approach is designed to deobfuscate scripts. In other words, the goal is to recover the original scripts. Therefore, scripts obfuscated by irreversible techniques, such as using random variables and function names and employing code virtualization (Reactor, 2003), cannot be addressed with this approach. Randomized variables and function names are used to reduce readability. They cannot be recovered but this does not affect the semantics of the scripts. For code virtualization obfuscation, dynamic approaches may be the only way to detect malicious scripts.

## 6.3 Evasion attacks

In this study, we propose a novel definition of obfuscation and a deobfuscation process, which reveal all the true commands hidden in the scripts. Attackers may disguise their malicious code as obfuscated parts so that malicious behaviors are executed during the deobfuscation process, and the final scripts look harmless. Although there is no such attack in the real world yet, attackers may design this kind of attack to evade deobfuscation and detection. A possible way to defend against this attack is to combine the processes of deobfuscation and detection. Specifically, we should traverse and deobfuscate the AST from bottom to top. Each possible command to be executed would be exposed during this process, and all these intermediate commands should be considered in detection.

# 7  Related works

## 7.1  Script-based malware detection

As shown in Table 8, malicious script detection methods fall into one of the following three categories:

1. Dynamic detection. JSAND (Cova et al., 2010) analyzes the runtime characteristic of suspicious scripts. Cujo (Rieck et al., 2010) further combines static and dynamic features to classify malicious samples with support vector machines (SVMs) and $N$-grams. CONAN (Xiong et al., 2022) aims to detect attacks via malicious dynamic behaviors and specifically treats scripting language interpreters. However, dynamic methods require extra runtime overhead and thus are inefficient for real-time detection and large-scale analysis.

2. Static detection. Prophiler (Canali et al., 2011) extracts multilayer features to identify benign scripts on web pages. Similarly, the method of Rusak et al. (2018) extracts static patterns from ASTs. ZOZZLE (Curtsinger et al., 2011) leverages ASTs

**Table 8  Comparison of script-based malware detection methods**

| Method | Light-weight | True positive | True negative | Semantic awareness | Anti-obfuscation |
|---|---|---|---|---|---|
| Dynamic detection | No | High | High | Yes | Yes |
| Static detection | Yes | Low | High | No | No |
| Obfuscation detection | Yes | Low | Low | No | Yes |
| Our approach | Yes | High | High | Yes | Yes |

for faster static signature matching. Both Hendler et al. (2018) and Rubin et al. (2019) used deep learning for malicious script detection. The former used character-level signatures and the latter used natural language processing (NLP) techniques for analysis. However, these methods are vulnerable to obfuscation and thus cannot be used in real-world attacks to detect obfuscated scripts. With the help of our deobfuscation method, the detection accuracy and coverage of these approaches can probably be improved significantly.

3. Obfuscation detection. Because obfuscation can help bypass nearly all static detection systems, researchers aim to detect obfuscation features to find malicious scripts. Several researchers (Kaplan et al., 2011; Jodavi et al., 2015; Aebersold et al., 2016) extracted different features to detect obfuscation for JavaScript, Bohannon (2017b) extracted 4098 features, and PowerDrive (Ugarte et al., 2019) generates regex expressions for PowerShell. However, there are three main limitations of these works: (1) They focus on entire obfuscated scripts and can thus be bypassed by partial obfuscation. (2) They assume that all obfuscated scripts are malicious, which is obviously a strong assumption and is not true in the real world (benign scripts can also be obfuscated for intelligence protection). (3) They use signatures to describe the features of obfuscated scripts but do not explain why these signatures are used; they are extracted from a set of samples and generality cannot be guaranteed.

## 7.2 Deobfuscation approaches

Several studies on deobfuscation have been conducted in recent years due to the popularity of "living off the land" and fileless attacks (Table 9). Specifically, PSDEM (Liu et al., 2018) manually generates specific signatures for each well-known obfuscation method, so it cannot deal with unknown obfuscation techniques and suffers from high false alarm rates when searching obfuscated script fragments (Li et al., 2019). PowerDrive (Ugarte et al., 2019) lists three common PowerShell obfuscation techniques and then manually generates regex expressions to deobfuscate them. It has the same disadvantages as PSDEM. PSDecode (R3MRUM, 2018) adopts a technique called method overriding to monitor the scripts to be executed, similar to Microsoft AMSI (Microsoft, 2019; Rubin et al., 2019).

However, dynamic methods require extra runtime overhead and are thus inefficient. Moreover, PSDecode and Microsoft AMSI cannot cover complicated obfuscated scripts according to our experiments (e.g., obfuscation scheme S1 and multilayer mix obfuscation). A simple example is shown in Table 10, which shows that AMSI cannot help obtain the original scripts in some common cases. In contrast, the approach proposed in this study has better generality, accuracy, and automation. First, our approach is more automatic, whereas PSDEM can deal only with obfuscation techniques that are analyzed manually. Second, our approach ensures that the deobfuscated scripts are consistent with

**Table 9  Comparison of representative deobfuscation approaches for script languages and our approach**

| Approach | Targeted language | Obfuscation detection accuracy | Recovery quality | Light-weight | Generality |
|---|---|---|---|---|---|
| PSDEM (Liu et al., 2018) | PowerShell | × | △ | ○ | × |
| PowerDrive (Ugarte et al., 2019) | PowerShell | × | △ | ○ | × |
| PSDecode (R3MRUM, 2018) | PowerShell | × | △ | ○ | × |
| JSDES (AbdelKhalek and Shosha, 2017) | JavaScript | × | △ | × | × |
| Lu and Debray (2012)'s approach | JavaScript | N/A | △ | × | × |
| Li et al. (2019)'s approach | PowerShell | △ | △ | △ | ○ |
| Our approach | PowerShell | ○ | ○ | ○ | ○ |

○, best; △, good; ×, bad; N/A, not available

**Table 10  A simple example of the limitation of AMSI\***

| Original | Obfuscated | AMSI log |
|---|---|---|
| tasklist | ('{1}{0}' -f 'list', 'task') \| Iex | ('{1}{0}' -f 'list', 'task') \| Iex tasklist |
| tasklist | .('{1}{0}' - f 'list', 'task') | .('{1}{0}' - f 'list', 'task') |

\*The test is done on Windows 10 v1906, and the obfuscation technique is commonly used in malicious scripts

the original scripts and are executable, whereas PS-DEM may locate obfuscated script fragments incorrectly and output meaningless strings. For other scripting languages, the method of AbdelKhalek and Shosha (2017) focuses only on function-level obfuscation and cannot deal with command-level obfuscation. Lu and Debray (2012) deobfuscated scripts by program slicing and dynamic execution, which has low code coverage and is limited by the instrumentation points. In contrast, our approach is generic and much faster than these dynamic methods due to our emulation execution strategy.

# 8  Conclusions

In this paper, we design and implement the first generic, effective, and lightweight deobfuscation approach for PowerShell scripts. To find the obfuscated fragments of scripts accurately, unlike existing works (AbdelKhalek and Shosha, 2017; Liu et al., 2018; Li et al., 2019; Ugarte et al., 2019), we propose a novel method based on the impact on ASTs. The results show that the new method performs better than the existing methods with almost no false positives, and that it can cover complicated, multilayer, and even unknown obfuscations. To recover the obfuscated scripts, we propose emulation-based recovery at the AST level. This method can recover the scripts perfectly, especially at the script-block level, for which it achieves 100% similarity. We mine 31 newly identified OOA rules automatically with our detection system. Based on the evaluation of more than 6483 obfuscated PowerShell scripts, it is proven that our method is not only efficient and effective but also generic. With the help of our deobfuscation system, the detection rates of existing malicious script detectors, including Windows Defender and the detectors in VirusTotal, are all significantly improved. Our detection system performs better than both Windows Defender and VirusTotal. The OOA rules extracted by our detector can be used directly by other analyzers and detectors to improve semantic-aware analysis and detection rates.

## Contributors

## Compliance with ethics guidelines

## References

AbdelKhalek M, Shosha A, 2017. JSDES: an automated de-obfuscation system for malicious JavaScript. Proc 12th Int Conf on Availability, Reliability and Security, p.1-13. https://doi.org/10.1145/3098954.3107009

Ackerman G, Cole R, Thompson A, et al., 2018. OVER-RULED: Containing a Potentially Destructive Adversary. https://bit.ly/2tSUacy [Accessed on Aug. 8, 2020].

Acornjs, 2013. Acorn. https://bit.ly/2BPzkyw [Accessed on Aug. 8, 2020].

Aebersold S, Kryszczuk K, Paganoni S, et al., 2016. Detecting obfuscated JavaScript using machine learning. 11th Int Conf on Internet Monitoring and Protection, p.11-17.

Ahl I, 2017. Threat Research: Privileges and Credentials: Phished at the Request of Counsel. https://bit.ly/2RaIk5o [Accessed on Aug. 8, 2020].

AST Explorer, 2015. AST Explorer. https://astexplorer.net/ [Accessed on Aug. 8, 2020].

Barak B, Goldreich O, Impagliazzo R, et al., 2012. On the (im)possibility of obfuscating programs. *J ACM*, 59(2):6. https://doi.org/10.1145/2160158.2160159

Bohannon D, 2016. Invoke-Obfuscation. https://bit.ly/2TIEwLN [Accessed on Aug. 8, 2020].

Bohannon D, 2017a. ObfuscatedEmpire–Use an Obfuscated, In-memory PowerShell C2 Channel to Evade AV Signatures. https://bit.ly/36UVYjC [Accessed on Aug. 8, 2020].

Bohannon D, 2017b. PowerShellObfuscation Detection Framework. https://bit.ly/2RhakUP [Accessed on Aug. 8, 2020].

Borgelt C, 2005. An implementation of the FP-growth algorithm. Proc 1st Int Workshop on Open Source Data Mining: Frequent Pattern Mining Implementations, p.1-5. https://doi.org/10.1145/1133905.1133907

Canali D, Cova M, Vigna G, et al., 2011. Prophiler: a fast filter for the large-scale detection of malicious web pages. Proc 20th Int Conf on World Wide Web, p.197-206. https://doi.org/10.1145/1963405.1963436

Candid W, 2016. The Increased Use of PowerShell in Attacks. https://symc.ly/2NmazwO [Accessed on Aug. 8, 2020].

Christodorescu M, Jha S, Seshia SA, et al., 2005. Semantics-aware malware detection. Proc IEEE Symp on Security and Privacy, p.32-46. https://doi.org/10.1109/SP.2005.20

Cova M, Kruegel C, Vigna G, 2010. Detection and analysis of drive-by-download attacks and malicious JavaScript code. Proc 19th Int Conf on World Wide Web, p.281-290. https://doi.org/10.1145/1772690.1772720

CrowdStrike, 2014. Free Automated Malware Analysis Service. https://bit.ly/36SUUgd [Accessed on Aug. 8, 2020].

CrowdStrike, 2018. Who Needs Malware? How Adversaries Use Fileless Attacks to Evade Your Security. https://bit.ly/2HZB23i [Accessed on Aug. 8, 2020].

Curtsinger C, Livshits B, Zorn B, et al., 2011. ZOZZLE: fast and precise in-browser JavaScript malware detection. Proc 20th USENIX Conf on Security, p.33-48.

Diggs R, 2017. Pulling Back the Curtains on EncodedCommand PowerShell Attacks. https://bit.ly/30jVNMr [Accessed on Aug. 8, 2020].

EmpireProject, 2015. Empire Is a PowerShell and Python Post-Exploitation Agent. https://bit.ly/36P13du [Accessed on Aug. 8, 2020].

FOLDOC, 1994. Free On-line Dictionary of Computing: Abstract Syntax Tree. https://foldoc.org/abstract+syntax+tree [Accessed on Aug. 8, 2020].

Fredrikson M, Jha S, Christodorescu M, et al., 2010. Synthesizing near-optimal malware specifications from suspicious behaviors. Proc IEEE Symp on Security and Privacy, p.45-60. https://doi.org/10.1109/SP.2010.11

Google, 2004. VirusTotal. https://bit.ly/3a3Pfpz [Accessed on Aug. 8, 2020].

Google, 2011. Traceur-Compiler. https://bit.ly/2BW2hZP [Accessed on Aug. 8, 2020].

Hendler D, Kels S, Rubin A, 2018. Detecting malicious PowerShell commands using deep neural networks. Proc Asia Conf on Computer and Communications Security, p.187-197. https://doi.org/10.1145/3196494.3196511

Hidayat A, 2012. ECMAScript Parsing Infrastructure for Multipurpose Analysis. https://esprima.org/ [Accessed on Aug. 8, 2020].

Jodavi M, Abadi M, Parhizkar E, 2015. JSObfusDetector: a binary PSO-based one-class classifier ensemble to detect obfuscated JavaScript code. Proc Int Symp on Artificial Intelligence and Signal Processing, p.322-327. https://doi.org/10.1109/AISP.2015.7123508

Kachalov T, 2016. JavaScript-Obfuscator. https://bit.ly/3cSvP7a [Accessed on Aug. 8, 2020].

Kannumittal, 2018. Difference b/w a Programming & Scripting Language. https://www.codingninjas.com/blog/2018/12/08/difference-between-a-programming-language-and-a-scripting-language/

Kaplan S, Livshits B, Zorn B, et al., 2011. "NOFUS: Automatically Detecting" String.fromCharCode(32) "ObFuSCateD" to LowerCase() "JavaScript Code". Technical Report MSR-TR 2011-57. Microsoft Research.

Koschke R, Falke R, Frenzel P, 2006. Clone detection using abstract syntax suffix trees. Proc 13th Working Conf on Reverse Engineering, p.253-262. https://doi.org/10.1109/WCRE.2006.18

Li ZY, Chen QA, Xiong CL, et al., 2019. Effective and lightweight deobfuscation and semantic-aware attack detection for PowerShell scripts. Proc ACM SIGSAC Conf on Computer and Communications Security, p.1831-1847. https://doi.org/10.1145/3319535.3363187

Liu C, Xia B, Yu M, et al., 2018. PSDEM: a feasible deobfuscation method for malicious PowerShell detection. Proc IEEE Symp on Computers and Communications, p.825-831. https://doi.org/10.1109/ISCC.2018.8538691

Lu G, Debray S, 2012. Automatic simplification of obfuscated JavaScript code: a semantics-based approach. Proc IEEE 6th Int Conf on Software Security and Reliability, p.31-40. https://doi.org/10.1109/SERE.2012.13

Maniar V, 2018. PowerShell-RAT. https://bit.ly/2uOD7ZH [Accessed on Aug. 8, 2020].

Mateas M, Montfort N, 2005. A box, darkly: obfuscation, weird languages, and code aesthetics. Proc 6th Digital Arts and Culture Conf, p.144-153.

Microsoft, 2014. Submit a File for Malware Analysis—Microsoft Security Intelligence. https://bit.ly/2TgVYXo [Accessed on Aug. 8, 2020].

Microsoft, 2019. Antimalware Scan Interface (AMSI). https://bit.ly/3hHhXBJ [Accessed on Aug. 8, 2020].

Mishoo, 2015. UglifyJS. https://bit.ly/30wOWkM [Accessed on Aug. 8, 2020].

MITRE, 2015. MITRE ATT&CK. https://attack.mitre.org/ [Accessed on Aug. 8, 2020].

MITRE, 2020. Technique: PowerShell-MITRE ATT&CK™. https://bit.ly/36SVSsR [Accessed on Aug. 8, 2020].

PowerShellMafia, 2012. PowerSploit: a PowerShell Post-Exploitation Framework—PowerShellMafia/PowerSploit. https://bit.ly/36STQJ9 [Accessed on Aug. 8, 2020].

R3MRUM, 2018. PowerShell Script for Deobfuscating Encoded PowerShell Scripts: R3mrum/PSDecode https://github.com/R3MRUM/PSDecode [Accessed on Aug. 8, 2020].

Reactor NET, 2003. Code Virtualization. https://www.eziriz.com [Accessed on Aug. 8, 2020].

Rieck K, Krueger T, Dewald A, 2010. Cujo: efficient detection and prevention of drive-by-download attacks. Proc 26th Annual Computer Security Applications Conf, p.31-39. https://doi.org/10.1145/1920261.1920267

Rubin A, Kels S, Hendler D, 2019. AMSI-based detection of malicious PowerShell code using contextual embeddings. https://arxiv.org/abs/1905.09538

Rusak G, Al-Dujaili A, O'Reilly UM, 2018. AST-based deep learning for detecting malicious PowerShell. Proc ACM SIGSAC Conf on Computer and Communications Security, p.2276-2278. https://doi.org/10.1145/3243734.3278496

Samratashok, 2020. What Is PowerShell? https://bit.ly/3f8U5DS [Accessed on Aug. 8, 2020].

Scraper W, 2019. Web Scraper. https://www.webscraper.io/ [Accessed on Aug. 8, 2020].

ShapeSecurity, 2015. Shift-parser-js. https://bit.ly/3fe0HRj [Accessed on Aug. 8, 2020].

Shen YD, Zhang Z, Yang Q, 2002. Objective-oriented utility-based association mining. Proc IEEE Int Conf on Data Mining, p.426-433. https://doi.org/10.1109/ICDM.2002.1183938

Symantec, 2018. Security Center White Papers | Symantec. https://symc.ly/2TlKphr [Accessed on Aug. 8, 2020].

Tobias W, 2018. New Obfuscation Modes. https://bit.ly/2FJhJae [Accessed on Aug. 8, 2020].

Ugarte D, Maiorca D, Cara F, et al., 2019. PowerDrive: accurate de-obfuscation and analysis of PowerShell malware. Proc 16th Int Conf on Detection of Intrusions and Malware, and Vulnerability Assessment, p.240-259. https://doi.org/10.1007/978-3-030-22038-9_12

Wueest C, Anand H, 2017. ISTR Living off the Land and Fileless Attack Techniques. https://symc.ly/2FP6v3X [Accessed on Aug. 8, 2020].

Wueest C, Stephen D, 2016. The Increased Use of PowerShell in Attacks. https://symc.ly/35Qj1ef [Accessed on Aug. 8, 2020].

Xiong CL, Zhu TT, Dong WH, et al., 2022. Conan: a practical real-time APT detection system with high accuracy and efficiency. *IEEE Trans Depend Sec Comput*, 19(1):551-565.
https://doi.org/10.1109/TDSC.2020.2971484

Xu W, Zhang FF, Zhu SC, 2012. The power of obfuscation techniques in malicious JavaScript code: a measurement study. Proc 7$^{th}$ Int Conf on Malicious and Unwanted Software, p.9-16.
https://doi.org/10.1109/MALWARE.2012.6461002

Ye YF, Wang DD, Li T, et al., 2008. An intelligent PE-malware detection system based on association mining. *J Comput Virol*, 4(4):323-334.
https://doi.org/10.1007/s11416-008-0082-4