# AST2Vec: A Robust Neural Code Representation for Malicious PowerShell Detection

Han Miao[1,2], Huaifeng Bao[1,2], Zixian Tang[1,2], Wenhao Li[1,2], Wen Wang[1(✉)], Huashan Chen[1], Feng Liu[1,2], and Yanhui Sun[3]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
wangwen@iie.ac.cn
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
[3] China Assets Cybersecurity Technology Co., Ltd., Beijing, China

**Abstract.** In recent years, PowerShell has become a commonly used carrier to wage cyber attacks. As a script, PowerShell is easy to obfuscate to evade detection. Thus, they are difficult to detect directly using traditional anti-virus software. Existing advanced detection methods generally recover obfuscated scripts before detection. However, most deobfuscation tools can not achieve precise recovery on obfuscated scripts due to emerging obfuscation techniques. To solve the problem, we propose a robust neural code representation method, namely AST2Vec, to detect malicious PowerShell without de-obfuscating scripts. 6 Abstract Syntax Tree (AST) recovery-related statement nodes are defined to identify obfuscated subtrees. Then AST2Vec splits the large AST of entire PowerShell scripts into a set of small subtrees rooted by these 6 types of nodes and performs tree-based neural embeddings on all extracted subtrees by capturing lexical and syntactical knowledge of statement nodes. Based on the sequence of statement vectors, a bidirectional recursive neural network (Bi-RNN) is modeled to leverage the context of statements and finally produce vector representation of scripts. We evaluate the proposed method for malicious PowerShell detection through extensive experiments. Experimental results indicate that our model outperforms the state-of-the-art approaches.

**Keywords:** PowerShell scripts · malware detection · abstract syntax trees · representation learning · Bi-RNN

## 1 Introduction

PowerShell, the scripting language and shell framework installed by default on most Windows computers, is becoming a favored attack tool by malware. It provides access to most major functions of the operating system, which makes it an ideal candidate for many abused purpose such as reconnaissance, gaining


M. Yung et al. (Eds.): SciSec 2023, LNCS 14299, pp. 207–224, 2023.
https://doi.org/10.1007/978-3-031-45933-7_13

persistence in the attacked system, communicating with a command and control server or downloading a payload. What's more, as a script, it is easy to obfuscate and difficult to detect with traditional security tools. According to a report from Symantec, over 95% of scripts using PowerShell were found to be malicious [10].

Many methods have been proposed to detect malicious PowerShell scripts. These methods can be categorized into two types: dynamic detection and static detection. Dynamic analysis often executes samples in an isolated environment and records their behaviors. However, there are many techniques to circumvent this method, such as sandbox awareness, which can hinder the execution of dynamic analysis. Static analysis often extracts static features such as strings for further detection, including two representative methods: 1) rule-based methods, which identify if the script matches one of the malware rules among the library collected in advance; 2) learning-based methods, which train machine learning or deep learning models using different static features, such as textual, token and AST node features [1,2,6,19] and verify new scripts based on this model. However, these approaches can be evaded by obfuscated scripts because obfuscation can easily change features mentioned above.

To tackle the issues, PowerShell deobfuscation work have been proposed. PSDEM [20], PSDecode [21] and PowerDrive [22] and PowerDecode [23] design a set of regular expressions to match obfuscated script pieces. Li et al. [4] identify obfuscated script pieces using a machine learning based classifier and conduct recovery through a simulation-based way. Invoke-Deobfuscation [3] utilizes recoverable nodes on AST to identify obfuscated pieces and implements variable tracing to mitigate the challenge above. However, all these approaches have some limitations. Regex expression based approaches are efficient, yet they often encounter syntax errors and semantics inconsistency due to lacking context and wrong replacement. Simulation-based approaches like Invoke-Deobfuscation obtain better results than regex expression based ones, but time-consuming and failed to deal with some sophisticated cases.

Thus, detecting malicious PowerShell scripts without deobfuscation can be considered as a promising approach to solve the problem. Intuitively, the idea is feasible for multiple reasons.

- In analogy to encrypted traffic analytics, many state-of-the-art works in this field like ETA [5] are typical content-independent approaches that only consider the observable metadata in structure level of traffic fragments and ignore the encrypted data information. Another similar task is software clone detection. Code obfuscation techniques are also used to prevent software clone detection. Some structure-based methods like ASTNN [7] have been proposed to solve the problem and achieve good results. These two works inspired us that structure may contain a large amount of useful information, which is not inferior to lexical information. We can formulate malicious PowerShell scripts detection as a similar task.
- Most new PowerShell scripts recycle large chunks of source code from existing scripts with some changes and additions. These scripts are similar in structure and functions, and just different in specific parameters. If the training data

were sufficient to contain malicious obfuscation scripts, the trained model would identify the similar malicious samples, no matter whether these samples were obfuscated or not.

In this paper, we propose a robust approach for representing PowerShell scripts that do not have to be deobfuscated, called AST2Vec, which splits the large AST of entire PowerShell scripts into a set of small subtrees and performs tree-based neural embeddings on all extracted subtrees. The contributions of this paper can be summarized as follows:

- We propose an efficient, effective and robust neural PowerShell scripts representation, which can capture the code fragments about recovering obfuscated pieces.
- Based on the new neural representation method, we design the first malicious PowerShell detection system without deobfuscation.
- The result shows that detection based on AST2Vec representation was enough to achieve a 3-fold cross-validation accuracy of 96.7% on obfuscated dataset.

The rest of this paper is organized as follows: Related work and preliminaries are introduced in Sect. 2 and Sect. 3, respectively. Section 4 presents the methodology. Section 5 describes the implementation and results of experiments. Discussion is presented in Sect. 6. Finally, we conclude the paper in Sect. 6.

## 2 Related Work

### 2.1 Obfuscation Techniques

Obfuscation techniques are often used by malware-writers to obfuscate their code and hinder static analysis. Compression, encryption, and encoding are some of the most common obfuscation methods used by threat actors. Multiple methods are often used in tandem to evade a wider variety of cyber security tools at the initial point of intrusion.

**String-Based Obfuscation.** String based obfuscation techniques typically include encoding and encryption [13–15], which are common in scripts and Java code. It rewrites the names of various elements of the code, such as variables, functions, and classes, into meaningless names. This greatly reduces the readability of the code, making it impossible for readers to guess its purpose based on the name.

**Logical Obfuscation.** Logical obfuscation is another type of obfuscation technique targeted at disrupting the control flow and the data flow, which changes the code into a functionally equivalent but less understandable form by rewriting

parts of the logic in the code. It is commonly used to obfuscate binary samples. LLVM is a tool for anti-sample reverse analysis [29]. It uses multiple optimizers to achieve a variety of logical obfuscation effects, such as code control flow flattening, false control flow, instruction replacement, etc. Analyzing samples that are confused by logic is very difficult.

### 2.2 Detection of Obfuscated Malware

In recent years, static analysis techniques [20–23], dynamic analysis techniques [30], as well as machine learning [1,4] and deep learning [2,19,24] have been widely used in malicious code detection and have achieved good results. However, detecting obfuscated malicious code remains a major challenge, as obfuscation techniques can make logically and functionally similar code appear completely different. Therefore, additional work is needed to assist traditional detection methods. Nicola Ruaro et al. [28] proposed a technology based on Symbolic Execution to disambiguate advanced malicious Excel files. Overall, the task of detecting malicious code without resolving confusion is very difficult.

### 2.3 Deobfuscation Techniques

**Dynamic Analysis Based Deobfuscation Techniques.** Dynamic analysis often executes samples in an isolated environment and records their behaviors [30]. However, there are many techniques that can hinder the execution of dynamic analysis, such as sandbox detection. Besides, benign scripts and malicious scripts quite frequently exhibit the same behaviors and functionality so without supplemental context or meta-data, it's hard to discern their intent on the surface by dynamic analysis.

**Statistic Analysis Based Deobfuscation Techniques.** Static analysis identifies obfuscated data and the corresponding recovery algorithm, which is usually very difficult. Common deobfuscation techniques based on statistic analysis can be divided into two types: regex expression based and AST based. Regex expression based tools process scripts at the level of strings and ignore the syntax of script pieces so that they cannot identify obfuscation pieces precisely. Li et al. [4]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 [3] utilizes recoverable nodes on AST to identify obfuscated pieces and implements variable tracing to mitigate the challenge above. However, it does not consider the situation of loop obfuscation and custom encryption.

## 3 Preliminaries

### 3.1 Abstract Syntax Tree

Abstract Syntax Tree (AST) is a hierarchical tree-like structure that captures the structure of a program's syntax, without including details such as formatting,

comments, or other non-essential information. The nodes in the tree represent the program's constructs, such as expressions, statements, functions, classes, and other language-specific constructs. As illustrated in Fig. 1(b), nodes of an AST are corresponding to constructs and symbols of the PowerShell script. The high-level abstraction makes ASTs widely used in software engineering and programming language analysis fields.

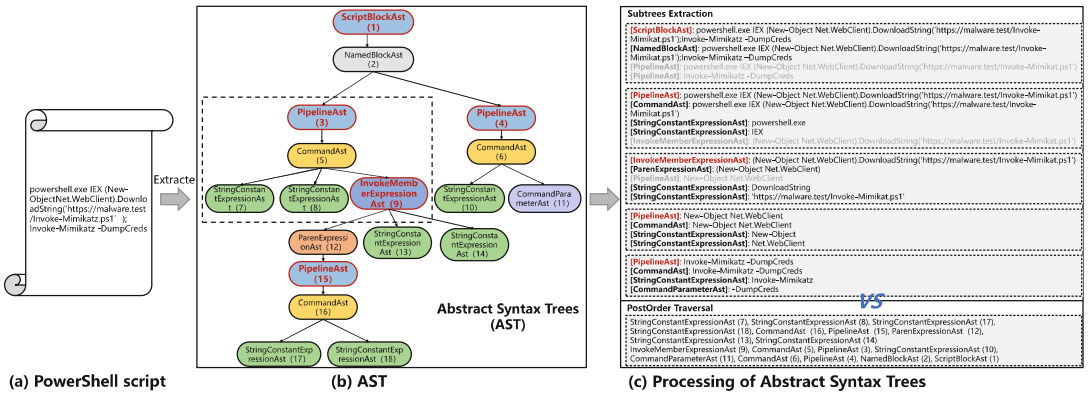


**Fig. 1.** An example of AST extraction.

Some studies [19,24] use ASTs in machine learning or deep learning based methods for malicious PowerShell detection by traversing or encoding ASTs as an entire or extracting statistical characteristics of trees. We shall be using ASTs as well but split the large AST of one code fragment into a set of small trees at the statement level for the next steps. The lower part of Fig. 1(c) illustrates the post order traversal of AST to get a sequence which destroys the original syntactic structure of source code. The upper part represents the newly proposed preprocessing method in this paper. Details will be covered in Sect. 3.

### 3.2 Tree-Based Neural Networks

Recently, many methods have proposed to take ASTs as input of Tree-based Neural Networks (TNNs) [7,8,18]. Given a tree, TNNs learns its vector representation by recursively computing node embeddings in a bottom-up way. Tree-based CNN (TBCNN) [8] and Tree-based Recursive Neural Networks (ASTNN) [7] are representative works in the field of tree-based neural networks.

**Tree-Based Convolutional Neural Network.** TBCNN is a tree-based convolutional neural network which uses CNN over tree structures for supervised learning such as source code classification [8]. The most important module of it is an AST-based convolutional layer, which applies a set of fixed-depth feature detectors by sliding over entire ASTs. This procedure can be formulated by:

$$y = tanh(\sum_{i=1}^{n} W_{conv,i} \cdot x_i + b_{conv})$$

where $x_1, \cdots, x_n$ are the vectors of nodes within each sliding window, $W_{counv,i}$ is the parameter matrices and $b_conv$ is the bias. TBCNN adopts a bottom-up encoding layer to integrate some global information for improving its localness. Although nodes in the original AST may have more than two children, TBCNN treats ASTs as continuous full binary trees because of the fixed size of convolution.

**Tree-Based Recursive Neural Network.** ASTNN is a generalization of RNNs to model tree-structured topologies. Different from standard RNN, ASTNN [7] recursively combines current input with its children states for state updating across the tree structure. Zhang et al. [7] uses ASTNN to learn representations of code fragments for clone detection, where code fragments are parsed to ASTs. To deal with the variable number of children nodes, a dynamic batching algorithm was proposed to put all possible children nodes with the same positions to one group. After a bottom-up way of computation, the root node vectors of ASTs are used to represent the code fragments.

## 4 Methodology

### 4.1 Framework

In this section, we introduce our malicious PowerShell detection approaches based on AST-based neural representation method (AST2Vec). The overall framework of our AST2Vec based detection is shown in Fig. 2. At a high level, the process of detection can be divided into three stages.

1) **Extraction of ASTs.** During the first stage, we parse a PowerShell script into an AST.
2) **Representation learning.** Then, we design a preorder traversal algorithm to split each AST to a sequence of subtrees, as illustrated in Fig. 1(c). All subtrees are encoded by the Subtree Encoder to vectors, denoted as $e_1, \cdots, e_t$. We then use Bidirectional Gated Recurrent Unit [16] (Bi-GRU), to model the context of the subtrees. The hidden states of Bi-GRU are sampled into a single vector by pooling, which is the representation of the scripts. Details of AST2Vec are illustrated in Fig. 3.
3) **Classifier training.** Finally, we employ the embedding as a first layer inputs in a deep neural network trained (using the labeled instances of the training set) to detect malicious PowerShell code.

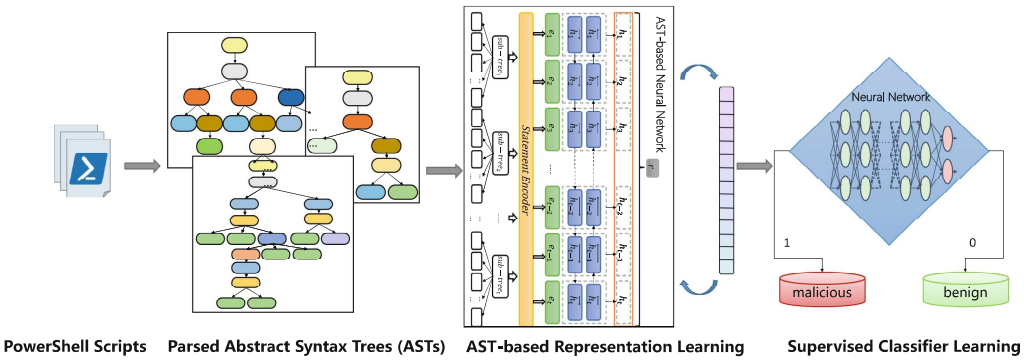


**Fig. 2.** The general framework of AST-based neural network for malicious PowerShell detection.

### 4.2 Splitting ASTs and Constructing Subtree Sequences

**The Granularity of Subtrees.** We adopt Microsoft's official library *System.Management.Automation.Language* to obtain ASTs for PowerShell. There are 71 types of PowerShell's AST nodes in total, such as PipelineAst, CommandAst, CommandExpressionAst, etc. As illustrated in Fig. 1(b), the parser returns an AST with a ScriptBlockAst type of root. A typical script with sizes of several Kilobytes can have thousands of nodes in AST, which means thousands of subtrees and thus makes it time-consuming to consider all subtrees.

To solve the problem, we only consider the below 6 types of subtrees, subtrees roots of PipelineAst, UnaryExpressionAst, BinaryExpressionAst, ConvertExpressionAst, InvokeMemberExpressionAst and SubExpressionAst type. This choice has two advantages:

1) **Significantly reduce the count of subtrees.** Through our observation, the numbers of nodes for certain node types are typically increased during the obfuscation process. For example, string reordering will add several ParenExpressionAst nodes and StringConstantExpressAst nodes to AST, which makes the AST more redundant. Fortunately, nodes like PipelineAst, UnaryExpressionAst, BinaryExpressionAst, ConvertExpressionAst, InvokeMemberExpressionAst and SubExpressionAst are related to node recovery in obfuscated scripts [3]. We call these 6 types of subtrees, recovery-related statement subtrees. Subtrees rooted by these nodes may include abundant nodes like StringConstantExpressAst, which means extracting them as an entire subtree will significantly reduce the count of meaningless subtrees.
2) **Improve the interpretability of subtrees.** At the AST level, the deobfuscation process can be considered as converting subtrees in an obfuscated script to corresponding ones in the target script. So extracting subtrees rooted by recovery-related statement nodes and representing them as an entire will improve the interpretability of subtrees.

Based on this idea, we traverse the AST in a breadth-first manner to extract the subtrees rooted by recovery-related nodes mentioned above and push them into a stack for subsequent steps.

**Splitting ASTs and Constructing Subtree Sequences.** Firstly, we transform scripts to large ASTs by existing PowerShell parser. For each AST, the granularity of split is recovery-related statement mentioned before. Then, We extract the sequence of statement trees with a preorder traversal.

Given an AST $T$ and a set of recovery-related statement AST nodes $S$, each statement node $s \in S$ in $T$ corresponds to one recovery-related statement of scripts. We treat ScriptBlockAst as a special Statement node, thus $S = S \cup \{ScriptBlockAst\}$. As shown in Fig. 1, there're many nested statements in ASTs. To get subtrees without non-overlapping, we reserve the nested statements such as Pipeline and InvokeMemberExpression statements in the body of the parent tree and copy them as the header of descendants. All the descendants of statement node $s \in S$ is denoted by $D(s)$. For any $d \in D(s)$, if there exists one path from $s$ to $d$ through one node $p = d$, it means that the node $d$ is included by one statement in the body of statement $s$. We call node $d$ one substatement node of $s$. Then a subtree rooted by the statement node $s \in S$ is the tree consisting of node $s$ and all of its descendants, excluding its substatement nodes' descendants in $T$. For example, the subtree rooted by PipelineAst is surrounded by dashed lines in Fig. 1(b), which includes nodes such as "CommandAst", two "StringConstantExpressionAst" and "InvokeMemberExpressionAst" and excludes the nodes of "ParenExpressionAst" and its sibling and descendant nodes in the body. The node "InvokeMemberExpressionAst" is both in the body of PipelineAst statement tree and in the descendants as a header. In this way, one large AST can be split to a sequence of multi-way subtrees.

The splitting of ASTs is straightforward by a traverser which visits each node through the ASTs in a depth-first walk in preorder. Then, a constructor recursively creates a subtree to sequentially add to the subtree sequences. Such a practice guarantees that one new subtree is appended by the order of the scripts. Finally, we get the sequence of subtrees as the raw input of AST2Vec.

### 4.3 Encoding Multi-way Subtrees

In this section, we introduce how to encode recovery-related statements mentioned above on multi-way subtrees. There are two challenges: learning vector representations of statements and designing the batch processing algorithm to encode subtrees with different number of children nodes.

**Statement Vectors.** Given the subtrees, we design a Bi-RNN based statement encoder to learn vector representations of statements.

Firstly, we obtain all the symbols by postorder traversal of ASTs as the corpus for unsupervised representation training used word2vec [25]. The trained

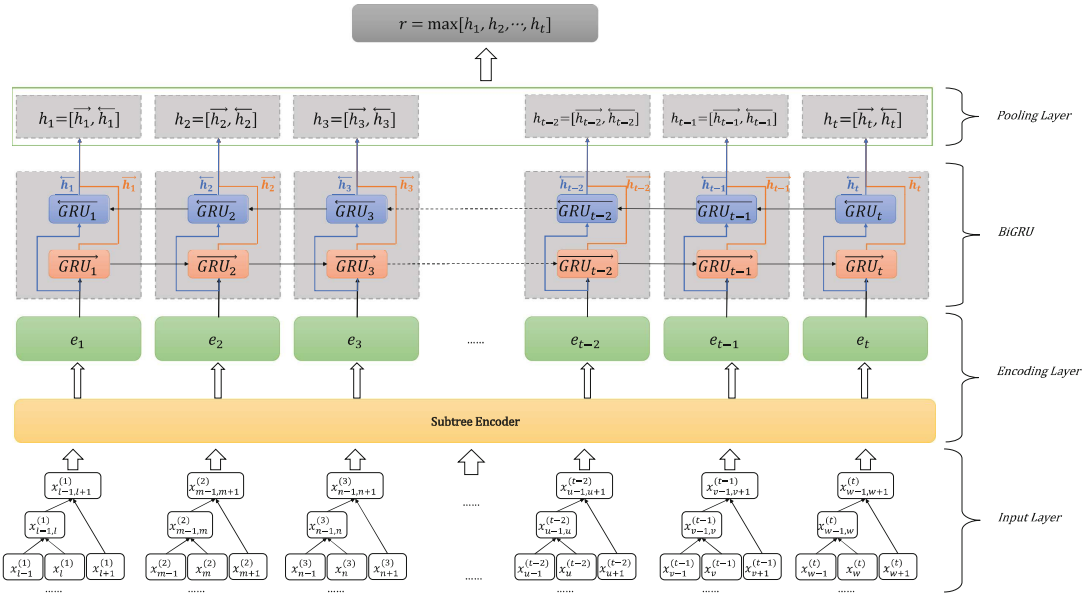


**Fig. 3.** The architecture of AST-based neural network.

embeddings of symbols are served as initial parameters in the subtree encoder, which are illustrated in the bottom in Fig. 3.

Taking the subtree rooted by the node of PipelineAST in Fig. 1(b) surrounded by dashed lines as an example, the encoder traverses the subtree and recursively takes the symbol of current node as new input to compute, together with the hidden states of its children nodes. This is illustrated in the lower two level in Fig. 4, in the subtree, the three children nodes–two StringConstantExpressionAst and a InvokeMemberExpressionAst enrich the meaning of CommandAst and the CommandAst transfer the merged information to the upper PipelineAST node.

Specifically, given a subtree $t$, let $n$ denote a non-leaf node and $C$ denote the count of its children nodes. At the beginning, with the pre-trained embedding parameters $W_\varepsilon \in \mathbb{R}^{|V|\times d}$ where $V$ is the vocabulary size and $d$ is the embedding dimension of symbols, the lexical vector of node $n$ can be obtained by:

$$v_n = W_\varepsilon^\top x_n \tag{1}$$

where $x_n$ is the one-hot representation of symbol $n$ and $v_n$ is the embedding. Next, the vector representation of node $n$ is computed by the following equation:

$$h = f(W_n^\top v_n + \sum_{i\in[1,C]} h_i + b_n) \tag{2}$$

where $W_n \in \mathbb{R}^{d\times k}$ is the weight matrix with encoding dimension $k$, $b_n$ is a bias term, $h_i$ is the hidden state for each child $i$, $h$ is the updated hidden state, and $f$ is the activation function such as $tanh$ or the identity function. We use the identity function in this paper. Similarly, we can recursively compute and optimize the vectors of all nodes in the subtree $t$. In addition, in order to determine the most

important features of the node vectors, all nodes are pushed into a stack and then sampled by the max pooling. That is, we get the final representation of the subtree and corresponding statement by Eq. 3, where $N$ is the number of nodes in the $subtree_t$.

$$e_t = [max(h_{i1}), \cdots, max(h_{ik})], i = 1, \cdots, N \tag{3}$$

These statement vectors can capture both lexical and statement-level syntactical information of the recovery-related statements.

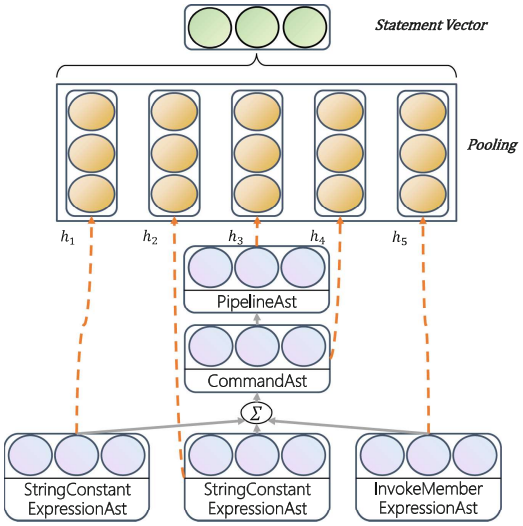


**Fig. 4.** The subtree encoder.

**Batch Processing.** There is a problem when batch processing on multiway subtrees, since the number of children nodes varies for the parent nodes in the same position of one batch. For example, given two parent nodes st1 with 3 children nodes and st2 with 2 children nodes in Fig. 5, directly calculating Eq. 2 for the two parents in one batch is impossible due to different $C$ values. To tackle this problem, we design an algorithm that dynamically processes batch samples.

Intuitively, although parent nodes have different number of children nodes, the algorithm can dynamically detect and put all possible children nodes with the same positions in groups, and then speed up the calculations of Eqs. 2 of each group in a batch way by leveraging matrix operations. An example is illustrated in Fig. 5.

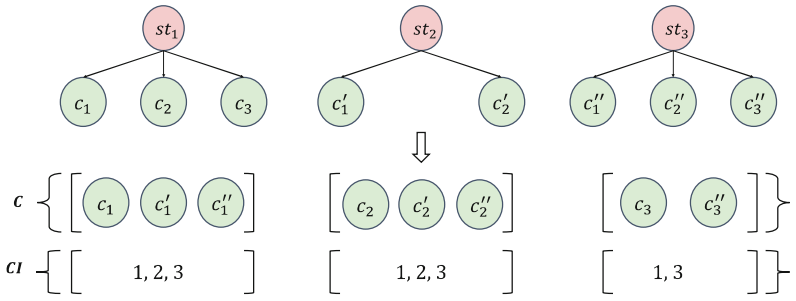


**Fig. 5.** An example of dynamically batching children nodes.

### 4.4 Representing the Sequence of Subtrees

After getting the sequences of subtrees vectors, we exploit GRU [17] to track the context of scripts.

Given a PowerShell script, suppose there are $T$ subtrees extracted from its AST and let $Q \in \mathbb{R}^{T\times k} = [e_1, \cdots, e_t, \cdots, e_T], t \in [1, T]$ denote the vectors of encoded subtrees in the sequence. At time $t$, the transition equations are as follows:

$$\begin{aligned} r_t &= f(W_r e_t + U_r h_{t-1} + b_r) \\ z_t &= f(W_z e_t + U_z h_{t-1} + b_z) \\ \tilde{h_t} &= tanh(W_h e_t + r_t \odot (U_h h_{t-1}) + b_h) \\ h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h_t} \end{aligned} \tag{4}$$

where $r_t$ is the reset gate to control the influence of previous state, $z_t$ is the update gate to combine past and new information, $\tilde{h_t}$ is the candidate state and used to make a linear interpolation together with previous state $h_{t-1}$ to determine the current state $h_t$. $W_r, W_z, W_h, U_r, U_z, U_h \in \mathbb{R}^{k\times m}$are weight matrices and $b_r$, $b_z$, $b_h$ are bias terms. $\odot$ denotes multiplication by elements. After iteratively computing hidden states of all time steps, the sequential context of these statements can be obtained.

In order to further enhance the capability of the recurrent layer for capturing the dependency information, we adopt a bidirectional GRU [16], where the hidden states of both directions are concatenated to form the new states as follows:

$$\begin{aligned} \overrightarrow{h_t} &= \overrightarrow{GRU}(e_t), t \in [1, T] \\ \overleftarrow{h_t} &= \overleftarrow{GRU}(e_t), t \in [T, 1] \\ h_t &= [\overrightarrow{h_t}, \overleftarrow{h_t}], t \in [1, T] \end{aligned} \tag{5}$$

Similar to the subtree encoder, the most important features of these states are then sampled by the max pooling or average pooling. Considering the importance of different statements are intuitively not equal, for example, InvokeMemberExpressionAst statements may contain more functional information than StringConstantExpressionAst, thus we use max pooling for capturing the most

important semantics by default. The model finally produces a vector $r \in R^{2m}$, which is treated as the vector representation of the PowerShell scripts.

### 4.5 Malicious PowerShell Detection

We use tree-based neural embeddings on ASTs to learn the malicious PowerShell detection model. Given the scripts vector $r$, we obtain the logits by $\hat{x} = W_l r + b_l$, where $W_l \in \mathbb{R}^{2m \times 2}$ is the weight matrix and $b_l$ is the bias term. We define the loss function as the widely used cross-entropy loss:

$$J(\Theta, \hat{x}, y) = \sum \left( -log \frac{exp(\hat{x}_y)}{\sum_j exp(\hat{x}_j)} \right) \tag{6}$$

## 5 Experiments and Results

### 5.1 Dataset Description

To evaluate our method, we create a collection of malicious and benign, obfuscated and nonobfuscated PowerShell samples. They come from all possible download sources that can have PowerShell scripts, e.g., GitHub, shared academic data, security blogs, etc. The overall information for datasets is listed in Table 1.

**Malicious Samples.** We get 3,346 obfuscated malicious PowerShell scripts from the author of Invoke-Deobfuscation [3] and download all 619 PowerShell instances from 'MalwareBazaar' website [12]. Besides, 4079 nonobfuscated malicious instances are obtained from unit42's blogs [11]. Finally, we removed invalid and duplicate PowerShell instances utilizing the syntax information and textual features of instances. After preprocessing, we ultimately get 8,005 malicious PowerShell samples.

**Benign Samples.** There are substantial PowerShell samples in PowerShellCorpus [26] created by Daniel Bohannon and most of them are benign. We select 8000 benign ones from the corpus by the help of VT [27]. To balance the proportion of obfuscated and nonobfuscated samples, we obfuscate half of them using popular obfuscation tools, e.g., Invoke-Obfuscation [13], PowerSploit [14], Empire [15], etc.

### 5.2 Experiment Settings

In order to obtain ASTs for PowerShell, we adopt Microsoft's official library *System.Management.Automation.Language* to parse PowerShell scripts to ASTs. We trained embeddings of AST nodes using word2vec [25] with Skip-gram algorithm and set the embedding size to be 128. The hidden dimension of subtree encoder and bidirectional GRU is 100. We set the batch size to 32 and a maximum of 20 epochs. The threshold is set to 0.5 for malicious PowerShell detection. We randomly divide the dataset into three parts, of which the proportions

**Table 1.** Overall information for datasets.

| Datasets | # Scripts | # Malicious instances | % Obfuscation |
|---|---|---|---|
| Invoke-Deobfuscation | 3,346 | 3,346 | 100 |
| MalwareBazaar | 580 | 580 | 51.72 |
| Unit42 | 4,079 | 4,079 | 0 |
| PowerShellCorpus(sub-fraction) | 8,000 | 0 | 50 |

are 60%, 20%, 20% for training, validation and testing. We use the optimizer AdaMax with learning rate 0.002 for training. All experiments are conducted on a computer with Intel Core i7-10750H CPU and 16 GB of memory for all experiments.

### 5.3 Evaluation on Detection Task

To evaluate the effectiveness of malicious PowerShell detection, we use the test accuracy metric, which computes the percentage of correct classifications for the test set. In this section, we compare the detection accuracy with state-of-the-art approaches in different groups including deobfuscation-based, simple AST features based and character-based approaches as follows. Experimental results are provided in Table 2 The best results are shown in bold.

**Deobfuscation-Based.** Li et al. [4] designed a light-weight deobfuscation approach for PowerShell scripts. Building upon this deobfuscation method, they further designed a semantic-aware PowerShell attack detection system. We reproduced the approach in our obfuscated and mixed datasets.

**Simple AST Features Based.** The detection method of Rusak et al. [19] used only two features: depth and number of nodes per PowerShell AST and was conducted on nonobfuscated scripts. This method didn't fully mining the potential information of abstract syntax trees. We reproduced the approach in our nonobfuscated, obfuscated and mixed datasets.

**Token-Based.** Hendler et al. [2] proposed a detection approach which apply token-based features for detection. The original design support several classifiers, and we only choose the combination of a 3-CNN and traditional 3-gram from which with the best results on their paper to reproduce. [6] was also proposed by Hendler et al. This method first gets deobfuscated scripts from AMSI, then extracts tokens and characters from scripts, uses Word2Vec for token embedding, and finally models convolutional neural networks and a Bi-LSTM to classify PowerShell scripts.

**Table 2.** Comparison with state-of-the-art detection approaches in accuracy.

| Detection approaches | Obfuscated scripts | Nonobfuscated scripts | Mixed Scripts |
|---|---|---|---|
| Li et al. [4] | – | 92.3% | 92.2% |
| Rusak et al. [19] | 0.0% | 85.7% | 9.6%2 |
| Hendler et al. [2] | 12.1% | 95.1% | 34.7% |
| ASMI-based [6] | 30.2% | 96.5% | 40.3% |
| AST2Vec | **96.7%** | **97.1%** | **96.4%** |

### 5.4 Results

Based on the evaluation on the above task, we aim to investigate the following research questions:

**RQ1: How does our approach perform in malicious PowerShell scripts detection?** In the task of malicious PowerShell scripts detection, the samples are strictly balanced among the malicious and benign classes and our evaluation metric is the test accuracy. Experimental results are provided in Table 2. The best results are shown in bold.

As shown, except for deobfuscation-based method, both the AST-based and character-based detection approaches will be bypassed by obfuscation. Once these scripts are deobfuscated, our results show that these two previous approaches can achieve similar accuracy rates with our approach. However, we would like to note that since the features used by character-based detection approach are manually crafted, they can be more easily evaded compared to our structural-aware approach. To show this, we simply mix benign pieces into malicious samples at the granularity of script lines, which changes AST structure and character distributions without affecting the script behavior. In Table 2, we call them "mixed scripts". As shown, these mixed scripts can greatly decrease the accuracy rates of simple AST features based and character-based detection approaches, but cannot affect that of our approach.

**RQ2: What are the effects of different design choices for the proposed model?** We conduct experiments to study how different design choices affect the performance of the proposed model on the detection task. As shown in Table 3, we consider the following design choices:

- **Splitting granularities of ASTs.** There are many ways to split a large AST into different sequences of non-overlapping small trees. Besides our semantic level splitting, another possible way is to extract all nodes of the AST(AST-Nodes) as special "trees". In comparative experiment, we adopt post order traversal to extract all nodes of the AST. After splitting, the follow-up encoding and bidirectional GRU processing are the same as those in our model. We can see that our semantic-based splitting outperform extreme splitting

**Table 3.** Comparison between the proposed model and its design alternatives.

| Description | Malicious PowerShell detection |
|---|---|
| | F1 score (%) |
| Bi-LSTM instead of Bi-GRU | 97.2 |
| GRU instead of Bi-GRU | 95.4 |
| One-hot instead of Word2Vec | 96.0 |
| Removing Pooling-I | 96.6 |
| Removing Pooling-II | 94.2 |
| AST-Node(PostOrder traversal) | 92.2 |
| AST2Vec | **97.4** |

approaches of AST-Nodes. Our model achieves a better performance, as analyzed in Sect. 3, this is because it balances a good trade-off between the size of subtree and the richness of syntactical and semantic information.

– **Initial representation of nodes.** The word2vec [25] is used to learn unsupervised vectors of the symbols, and the trained embeddings of symbols are served as initial parameters in the subtree encoder. We extract all ASTs from the total PowerShell scripts and train embeddings on them. The qualitative results are summarized in a dendrogram in Fig. 4. It shows the relationships of embeddings with similar ones. Notably, the TryStatement and CatchClause node types are neighbors, as well as Command and CommandParameter. Afterward, we will use these embeddings as initial parameters in the subtree encoder to enrich the semantic information of en, since one would expect such commands to serve similar functions in scripts. There are 71 types of PowerShell's AST nodes in total. If replacing word2vec embeddings by one-hot representation, the results indicate that overall one-hot representation decrease the F1 score by 1.4%. This shows that the initial representation of AST nodes plays a significant role in detection.
– **GRU.** We use GRU in the recurrent layer of our proposed model by default. If replacing GRU by LSTM, the results indicate that overall LSTM has a slightly poor with GRU. We prefer GRU in our model since it achieves more efficient training.
– **Pooling layer.** In our model, we use the max pooling on subtrees in the statement encoder (Pooling-I) and the max pooling layer on the statement sequences after the recurrent layer (Pooling-II) as described in Sect. 3. We study whether the two pooling components affect the performance or not by removing them and directly using the last layer hidden states. From the Table 3, we can see that the pooling on statement sequences provides a comparatively significant performance boost, whereas pooling on subtrees matters little. This shows that different statements of the same scripts actually have different weights (Fig. 6).

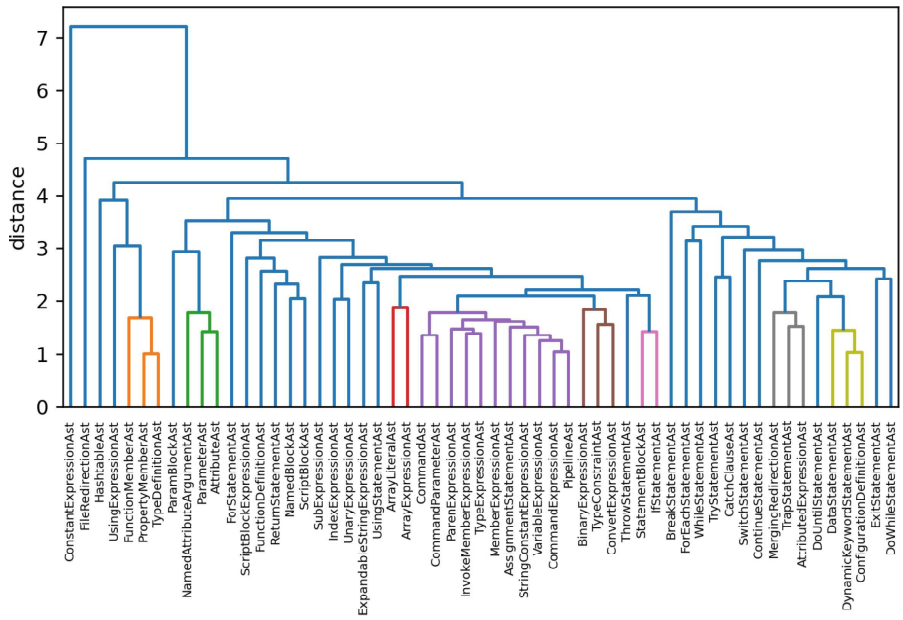


**Fig. 6.** Dendrogram of node types and their relationships in the Learning Node Representations experiment.

## 6 Discussion

### 6.1 Generality of Our Approach

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

Moreover, our method requires only a parser and an unmodified interpreter for the target language, both of which typically have official tools available. Therefore, the only extra work required to construct a detection system for a new language is to collect the samples of the obfuscated subtrees and train the attack detector.

### 6.2 Necessity of Obfuscation for Malicious PowerShell Detection

The results in Table 2 shows that our approach performs well on both obfuscated and nonobfuscated datasets. There are several reasons accounting for this. For one thing, most new PowerShell scripts recycle large chunks of source code from existing scripts with some changes and additions, whether the source code is obfuscated. These scripts are similar in structure and functions, and differ only in specific parameters. If these scripts are parsed to ASTs, they will have similar, even same patterns in ASTs. For another, our proposed approach learns vector representations of scripts fragments, which can capture the syntactical knowledge

and naturalness of semantic statements of the PowerShell well. Thus, If the training data are sufficient to contain malicious obfuscation scripts, the trained model would identify the similar malicious samples, no matter whether these samples are obfuscated or not.

## 7 Conclusion

In this paper, we design the first effective and feasible detection approach for malicious PowerShell scripts without deobfuscation. To address the key challenge of robustly representing the PowerShell scripts both obfuscated and nonobfuscated, we design a novel AST-based representation method that could capture the recovery-related statement at the level of subtrees. Building upon the representation method, we design the PowerShell attack detection system. Based on a collection of 16,005 PowerShell scripts, our detection model is shown to be both robust and effective.

**Acknowledgments.** This work was supported by the National Key R&D Program of China with No. 2021YFB3101402.

## References

1. Fang, Y., Zhou, X., Huang, C.: Effective method for detecting malicious PowerShell scripts based on hybrid features. Neurocomputing **448**, 30–39 (2021)
2. Hendler, D., Kels, S., Rubin, A.: Detecting malicious PowerShell commands using deep neural networks. In: Proceedings of the 2018 on Asia Conference on Computer and Communications Security (2018)
3. Chai, H., Ying, L., Duan, H., Zha, D.: Invoke-deobfuscation: AST-based and semantics-preserving deobfuscation for PowerShell scripts. In: 2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 295–306 (2022)
4. Li, Z., Chen, Q.A., Xiong, C., Chen, Y., Zhu, T., Yang, H.: Effective and lightweight deobfuscation and semantic-aware attack detection for PowerShell scripts. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (2019)
5. Blake, A., David, M.: Identifying encrypted malware traffic with contextual flow data. In: Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security, AISec 2016, pp. 35–46 (2016)
6. Hendler, D., Kels, S., Rubin, A.: AMSI-based detection of malicious PowerShell code using contextual embeddings. In: Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (2019)
7. Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X.: A novel neural source code representation based on abstract syntax tree. In: Proceedings of the 41st International Conference on Software Engineering, pp. 783–794. IEEE Press (2019)
8. Mou, L., Li, G., Zhang, L., Wang, T., Jin, Z.: Convolutional neural networks over tree structures for programming language processing (2015)
9. Mikolov, T., Karafiát, M., Burget, L., Cernock, J., Khudanpur, S.: Recurrent neural network based language model. In: Interspeech, Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September (2015)

10. ISTR Living off the land fileless attack techniques. https://www.symantec.com/content/dam/symantec/docs/security-center/whitepapers/istr-living-off-the-land-and-fileless-attack-techniques-en.pdf. Accessed 11 Apr 2023
11. karttoon, psencmds (2019). https://github.com/pan-unit42/iocs/commits/master/psencmds. Accessed 13 Dec 2019
12. MalwareBazaar. https://bazaar.abuse.ch/
13. Bohannon, D.: Invoke-obfuscation - powershell obfuscator. https://github.com/danielbohannon/Invoke-Obfuscation
14. Powersploit - a powershell post-exploitation framework. https://github.com/PowerShellMafia/PowerSploit
15. Empire - a PowerShell and python post-exploitation agent. https://github.com/EmpireProject/Empire
16. Tang, D., Qin, B., Liu, T.: Document modeling with gated recurrent neural network for sentiment classification. In: Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, pp. 1422–1432 (2015)
17. Bahdanau, D., Cho, K., Bengio, Y.: Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473 (2014)
18. Wei, H.-H., Li, M.: Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In: Proceedings of the 26th International Joint Conference on Artificial Intelligence, pp. 3034–3040. AAAI Press (2017)
19. Rusak, G., Al-Dujaili, A., O'Reilly, U.M.: AST-based deep learning for detecting malicious PowerShell. In: ACM CCS (2018)
20. Liu, C., Xia, B., Yu, M., Liu, Y.: PSDEM: a feasible de-obfuscation method for malicious PowerShell detection. In: IEEE ISCC (2018)
21. Psdecode - PowerShell script for deobfuscating encoded PowerShell scripts. https://github.com/R3MRUM/PSDecode
22. Ugarte, D., Maiorca, D., Cara, F., Giacinto, G.: PowerDrive: accurate de-obfuscation and analysis of PowerShell malware. In: Perdisci, R., Maurice, C., Giacinto, G., Almgren, M. (eds.) DIMVA 2019. LNCS, vol. 11543, pp. 240–259. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-22038-9_12
23. Malandrone, G.M., Virdis, G., Giacinto, G., Maiorca, D.: PowerDecode: a PowerShell script decoder dedicated to malware analysis. In: ITASEC (2021)
24. Gao, Y., Peng, G., Yang, X.: PowerShell malicious code family classification based on deep learning. J. Wuhan Univ. (Nat. Sci. Ed.) **68**(1), 8–16 (2022). https://doi.org/10.14188/j.1671-8836
25. Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J.: Distributed representations of words and phrases and their compositionality. In: Advances in Neural Information Processing Systems, pp. 3111–3119 (2013)
26. Bohannon, D., Holmes, L.: Revoke-Obfuscation: PowerShell Obfuscation Detection Using Science (2017). https://www.fireeye.com/blog/threatresearch/2017/07/revoke-obfuscation-powershell.html
27. VirusTotal. https://www.virustotal.com/
28. Ruaro, N., Pagani, F., Ortolani, S., Kruegel, C., Vigna, G.: SYMBEXCEL: automated analysis and understanding of malicious excel 4.0 macros. In: 43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, 22–26 May 2022, pp. 1066–1081. IEEE (2022)
29. LLVM. https://www.llvm.org/
30. Cozzi, E., Graziano, M., Fratantonio, Y., Balzarotti, D.: Understanding Linux malware. In: 2018 IEEE Symposium on Security and Privacy (S&P), pp. 161–175 (2018)

# Real-Time Aggregation for Massive Alerts Based on Dynamic Attack Granularity Graph

Haiping Wang[1,2], Binbin Li[1(✉)], Tianning Zang[1,2], Yifei Yang[1], Zisen Qi[1,2], Siyu Jia[1,2], and Yu Ding[1,2]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
{wanghaiping,libinbin}@iie.ac.cn
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
wanghaiping20@mails.ucas.ac.cn

**Abstract.** To perceive the overall cyber security situation of the target network, cyberspace situational awareness platforms collect billions of alerts from IDS, IPS, firewalls, probes, and third-party systems in real-time. It is urgent to aggregate these multi-source, massive, real-time, heterogeneous, dynamic, strong timeliness alerts to help analysts find valuable clues as quickly as possible. In this paper, we propose a novel real-time alert aggregation approach. It is based on a dynamic attack granularity graph model combined with a dynamic threshold update algorithm, which not only can effectively solve the redundancy problem of massive multi-source data, but also can extract valuable alert aggregation from the data flood. To evaluate our approach, we conduct experiments using the public datasets *Suricata* (81600 alerts) and a real 24-hour online dataset (1204753 alerts). We use evaluation metrics including *Aggregation Rate(AR), Simplicity Metric(SM) and Time Delay(TD)*, and select three common alert aggregation algorithms to perform a comparative test in a simulated real-time situation. The experiment shows that our approach achieves more than 98% aggregation rate, reduces data complexity by more than 82%, and has stronger robustness.

**Keywords:** Cyber Security Situation Awareness · Massive Alerts Aggregation · Real-time · Granularity Graph · Dynamic Threshold

## 1 Introduction

The existing cyber security situation awareness system needs to collect a large number of alert logs from network traffic monitoring devices such as IDS, IPS, firewalls, probes and third-party systems. The billions of alerts are multi-source, massive, real-time, heterogeneous, dynamic, and time-sensitive, which contain a lot of noise and redundant information. The valuable information is discrete and difficult to extract.

Existing solutions can be summarized into the following two aspects: solutions based on expert knowledge or experience, and algorithms based on machine


M. Yung et al. (Eds.): SciSec 2023, LNCS 14299, pp. 225–243, 2023.
https://doi.org/10.1007/978-3-031-45933-7_14