Computers & Security 154 (2025) 104441

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Full length article

Power-ASTNN: A deobfuscation and AST neural network enabled effective detection method for malicious PowerShell Scripts

Sanfeng Zhang [a,b,*], Shangze Li [a], Juncheng Lu [a], Wang Yang [a,b]

[a] School of Cyber Science and Engineering, Southeast University, Nanjing, China
[b] Key Laboratory of Computer Network and Information Integration (Southeast University), Ministry of Education, Nanjing, China

ARTICLE INFO

Keywords:
Malicious PowerShell script detection
Deobfuscation
AMSI memory dump
Abstract semantic tree neural network
Bi-GRU

ABSTRACT

PowerShell is frequently utilized by attackers in the realm of Windows system security, particularly in cyberattack activities such as information stealing, vulnerability exploitation, and password cracking. To evade detection, attackers often employ code obfuscation techniques on their scripts. Current detection solutions face challenges due to limited deobfuscation methods and a predominant focus on identifying static and local features. This limitation hinders the ability to capture fine-grained code features and long-distance semantic relationships, resulting in reduced robustness and accuracy. To address these issues, this paper presents a novel malicious script detection method, Power-ASTNN, which integrates deobfuscation and a tree neural network. Initially, the method utilizes AMSI memory dump to deobfuscate PowerShell scripts, yielding fully deobfuscated samples. Subsequently, a subtree splitting algorithm tailored for abstract syntax trees extracts fine-grained code features from subtree fragments. Finally, a two-layer neural network model encodes representations based on subtree node semantics and sequence semantics, effectively capturing the semantic characteristics of the code. Experimental results demonstrate the effectiveness of Power-ASTNN, achieving an accuracy of 98.87% on a self built dataset collected from multiple publicly available sources, while maintaining a low false negative rate and a high area under the curve (AUC) value exceeding 0.995. Furthermore, Power-ASTNN demonstrates superior detection performance against adversarial samples compared with existing detection models.

1. Introduction

In the context of Windows system security, PowerShell is a powerful first-party management tool that provides access to critical Windows components, enables privileged system-level operations, and supports fileless attacks. Cybercriminals frequently exploit PowerShell scripts for advanced persistent threat (APT) attacks, ransomware incidents (Palo Alto Networks, 2010a,b), and assaults on cloud infrastructures (Anon, 2014). Recent statistics indicate that 28.49% of attack activities over the past three years have involved obfuscated PowerShell command lines (Anon, 2015). To evade detection, attackers often obfuscate malicious scripts and inject adversarial perturbations, complicating detection efforts and undermining overall Windows system security.

Traditional detection methods for PowerShell scripts include signature-based matching using YARA rules and behavioral analysis through log records (Barr-Smith et al., 2021). However, these approaches often suffer from outdated rule bases that lag behind emerging threats and yield high false positive rates (Galloro et al., 2022). With advancements in artificial intelligence, researchers have proposed machine learning techniques that utilize syntax structure features (Rusak et al., 2018; Choi, 2020; Mimura and Tajiri, 2021; Fang et al., 2021; Song et al., 2021; Yang et al., 2023) and deep learning models based on code text analysis (Alahmadi et al., 2022; Hendler et al., 2018). Nonetheless, these methods frequently lack effective deobfuscation preprocessing, limiting their ability to reveal the syntax features of malicious scripts. Additionally, they struggle with fine-grained feature extraction, which constrains their detection performance.

Common pre-processing techniques such as Base64 decoding (Hendler et al., 2020), regular expression matching and replacement, and annotation and whitespace removal often fail to fully restore original code. Residual obfuscation alters the semantic features of the code, disrupting the arrangement of syntax tree nodes and hindering the neural network's ability to extract meaningful semantic information. This limitation adversely affects the accuracy and generalization capabilities of deep learning models (Gopinath and Sethuraman, 2023; Gaber et al., 2024). Furthermore, the altered structural features of the code impede the effective extraction of statistical characteristics, such as information entropy and high-frequency terms, thereby impacting the false alarm rate and robustness of machine learning models.

* Correspondence to: Southeast Univ, Sch Cyber Sci & Engn, Nanjing, PR China.
E-mail address: sfzhang@seu.edu.cn (S. Zhang).

https://doi.org/10.1016/j.cose.2025.104441
Received 4 October 2024; Received in revised form 16 February 2025; Accepted 11 March 2025
Available online 22 March 2025

Recent advancements in deobfuscation techniques include regular deobfuscation based on reverse engineering (Liu et al., 2018; Ugarte et al., 2019; Malandrone et al., 2021; R3MRUM, 2020) and solutions leveraging abstract syntax trees (ASTs) (Rose et al., 2024; Li et al., 2019; Xiong et al., 2022; Chai et al., 2022). However, the former struggles with complex, multi-layered obfuscation strategies, while the latter is inefficient due to the need for a global traversal of all syntax trees. Emerging research highlights the potential of memory dump techniques for code deobfuscation (Cheng et al., 2021, 2023; Li et al., 2022; Luo et al., 2023; Hou et al., 2024). Since the PowerShell interpreter executes only deobfuscated commands, the content of deobfuscated PowerShell scripts can be extracted using the Antimalware Scan Interface (AMSI) prior to execution. This content, however, often comprises a large number of various benign code snippets generated for deobfuscation. Accordingly, this paper further analyzes and extracts AMSI dump content to reveal more effective code semantics within PowerShell script samples.

In terms of representation learning and classification of PowerShell script features, existing research indicates that fine-grained feature extraction can be effectively achieved through the use of ASTs, thereby enhancing the code's feature representation and detection capabilities (Sun et al., 2023; Deng et al., 2022). Serialized ASTs facilitate context-based embedded learning of the sequences of nodes and subtrees, improving the completeness and accuracy of feature extraction. Structures of ASTs enable the analysis of features from sample syntax tree nodes and statements. By delving deeper into code structure and applying dimensionality reduction techniques, the time efficiency and robustness of feature extraction can be significantly improved. AST-based code representation learning techniques have demonstrated success in various domains, including code recommendation (Tao et al., 2022; Kara, 2023), code summarization (Lin et al., 2021; Shi et al., 2023), and vulnerability detection (Fass et al., 2018). However, advancements are still needed in the areas of efficient deobfuscation pre-processing and fine-grained feature extraction specifically for PowerShell scripts.

To address these challenges, we propose Power-ASTNN, a method for detecting malicious PowerShell scripts that leverages deobfuscation, fine-grained AST subtree decomposition, and tree neural networks. Initially, we utilize AMSI memory dump to deobfuscate PowerShell scripts, resulting in a fully deobfuscated representation for further analysis. Subsequently, we extract code features from fine-grained subtree code snippets using an abstract syntax tree-oriented subtree splitting algorithm. Finally, we employ a two-layer neural network model to represent the code at both the subtree node semantics level and sequence semantics level, effectively capturing the semantic relationships within the code.

Specifically, the main contributions of this paper are summarized as follows:

(1) AMSI memory dump and deobfuscation tool: We present a new tool that utilizes the AMSI interface to obtain deobfuscated PowerShell script content before execution. To enhance deobfuscation efficiency and ensure thorough results, we design a preprocessing method based on code subtree decomposition and merging, as well as a post-processing method employing double substitution to reveal more effective code semantics. This tool is available for the security community at https://404nfd.top.

(2) Fine-grained subtree decomposition method: We introduce a fine-grained subtree decomposition method tailored for fully deobfuscated PowerShell scripts. This method systematically traverses, extracts, and merges seven specific types of code subtrees, transforming complex scripts into structured sequences. Additionally, we implement a recursive algorithm to flatten nested statement trees within each subtree, facilitating semantic representation learning for the nested nodes in subsequent subtrees.

(3) Two-Layer code representation learning model: Our approach employs a two-layer model that integrates ASTs and tree neural networks for code feature extraction and semantic learning at the levels of AST nodes and node sequences. We apply a multi-head self-attention mechanism to AST subtree nodes to capture syntactic structural features, alongside a bidirectional recurrent neural network to learn long-distance code dependencies and contextual correlations. Compared to the ASTNN model (Zhang et al., 2019), our method processes each subtree in parallel, significantly enhancing the sample processing speed during training and inference.

(4) Optimized detection performance: Our method achieves a detection accuracy of 98.87% on a self built dataset collected from multiple publicly available sources, maintaining a very low false negative rate and a high AUC value (> 0.995). Notably, when confronted with adversarial and obfuscated samples, it surpasses existing detection models significantly.

2. Related work

2.1. Malicious PowerShell script detection

Existing methods for detecting malicious PowerShell scripts can be categorized into three primary approaches: knowledge-based rule matching, deep learning utilizing code text features, and deep learning focusing on syntactic structural features.

Knowledge-based rule matching methods identify malicious scripts by comparing their string features against a database of known malware signatures (Brengel and Rossow, 2021). Event log-based rule matching employs regular expressions to detect attack activities, such as memory injection and other malicious scripts (Bode et al., 2020). However, these methodologies depend on security experts for continuous knowledge extraction and rule updates, limiting their adaptability to evolving attack strategies. Furthermore, they struggle to effectively mitigate evasion techniques such as obfuscation and process bypass (Al-Saleh et al., 2013).

Deep learning approaches utilizing text features involve several stages, including sequence encoding, feature extraction, and model training. For instance, Hendler (Hendler et al., 2020) one-hot encodes the first 1000 characters of all samples and trains a four-layer convolutional neural network (CNN) using stochastic gradient descent for parameter optimization. Similarly, Mimura and Tajiri (2021) analyze word frequency across samples, creating a 200-dimensional frequency vector and employing XGBoost for classification. Alahmadi (Alahmadi et al., 2022) utilizes ASCII encoding for the first 1024 characters and trains a four-layer stacked denoising autoencoder, applying a sigmoid function to enhance feature learning. Although these unstructured feature extraction methods can automate the identification of sequential text features and yield high accuracy and AUC values, they are constrained by memory and computational resources, potentially leading to overfitting in long sequence scenarios and difficulties in processing real-time data streams.

For syntactic structural feature analysis, Rusak examined the number and depth of 64 PowerShell abstract syntax tree nodes, developing an embedded vector based on recursive nodes and employing random forests for code classification. Fang et al. (2021) proposed a hybrid feature model combining text features (e.g., information entropy, character length, URLs) and AST features (e.g., AST depth, node distribution) using the FastText model. This approach achieved optimal results with a random forest classifier after concatenating the various vector types. Although these syntax structure-based methods are not limited by input sample length and require less training time and cost, they face challenges such as susceptibility to obfuscated scripts, where benign codes may be misclassified as malicious, leading to elevated false positive rates. Additionally, the lack of fine-grained feature extraction diminishes the model's generalization ability and hinders accuracy improvements.

To enhance the model's ability for fine-grained semantic learning, a tree neural network (Alon et al., 2018) can be utilized to capture

Table 1
Selected 7 Subtrees

AST Type	Subtree Type Definition	Code Example
PipelineAST	Command and pipeline operations	Set-BuildEnvironment -Path
AssignmentStatementAST	Variable assignment statement	$RwQJ = sET Tm7Ao ("
FunctionDefinitionAST	Function definition body	FuNCTion SRhOQP {}
IfStatementAST	Conditional expression and statement	if (!(Get-Module WebAdministration))
ForeachStatementAST	Iterating collections or arrays	foreach($site in $Deployment)
WhileStatementAST	Repeated execution block	$whileAst = [WhileStatementAST]::new()
TryStatementAST	Error handling code	try { &('Enter build') 42} catch {${_}}

relationships between nodes and code sequences in long sequences. The AST serves as an abstract representation of the logical structure of source code, preserving its core elements. Tree neural networks can further extract features from the subtree nodes of the AST, enabling deeper code representation and effectively capturing contextual semantic associations. This paper presents a two-layer graph neural network designed to leverage both tree neural networks and AST features to learn the syntactic structure and sequential characteristics of PowerShell scripts.

2.2. PowerShell script deobfuscation

In the context of pre-processing for the deobfuscation of PowerShell scripts, existing detection schemes typically rely on basic techniques, including Base64 decoding, regular expression matching and replacement, and the removal of comments and whitespace. However, these methods often fall short of achieving adequate deobfuscation. For instance, regular expression matching can only address a limited range of obfuscation types; complex grammatical structures may result in syntax errors, hindering proper execution or interpretation of the scripts.

Given that many obfuscated script fragments contain self-recovering code, emulation execution proves effective for deobfuscation. For example, methodologies proposed by Li et al. (2019) and Chai et al. (2022). involve parsing the PowerShell script into an abstract syntax tree, executing the subtree containing the obfuscated statements, and ending the deobfuscation process based on variations in execution results. While these methods theoretically enhance the accuracy of obfuscated script restoration, they often incur long execution times due to process delays and variable tracing challenges. Additionally, failures in context tracing may further complicate deobfuscation efforts.

The Antimalware Scan Interface (AMSI) enables the retrieval of deobfuscated PowerShell scripts prior to actual execution. Recent studies have demonstrated the efficacy of memory dump techniques for deobfuscation. Although the AMSI memory dump technique was first employed in for script parsing and detection, it suffers from limitations in the pre-processing of obfuscated scripts and the post-processing of dumped scripts, resulting in suboptimal deobfuscation outcomes. The deobfuscation module presented in this paper also utilizes the AMSI memory dump technique but enhances the deobfuscation process through improved pre- and post-processing of the scripts, thereby significantly increasing its effectiveness.

3. Method

3.1. Overall structure

As illustrated in Fig. 1, the overall architecture of Power-ASTNN comprises an AMSI-based deobfuscation module, an AST subtree decomposition module, and an AST-based representation and classification module for PowerShell scripts. The AMSI-based deobfuscation module encompasses several operations, including AST pre-processing, merging of obfuscated code blocks, simulation execution, and post-processing of the dumped results. The AST subtree decomposition module conducts fine-grained decomposition of the AST, transforming complex scripts into a structured sequence of subtrees by traversing, extracting, and merging seven specific types of code subtrees. The feature learning and classification module employs a neural network model for feature extraction and semantic learning at two levels. It incorporates a multi-head self-attention mechanism to capture features from the nodes within the subtree sequence. Additionally, it utilizes a bidirectional recurrent neural network to analyze all AST subtree sequences, thereby capturing long-range code dependencies and learning contextual features of the code.

3.2. Deobfuscation based on AMSI memory dump

PowerShell obfuscated scripts must undergo a self-decoding operation to restore them to executable string code, commonly referred to as 'plain code' . The Antimalware Scan Interface provides a means to obtain this plain code, which is significantly less obfuscated than the original script. However, deobfuscation using AMSI directly presents challenges; for instance, the plain code discards unobfuscated portions, retaining only the code produced by the deobfuscation process, which complicates the direct and accurate restoration of the original script. Additionally, specific strategies are required to trace the content of obfuscated variables.

In the deobfuscation phase utilizing AMSI, Power-ASTNN implements three critical steps: AST-Parser pre-processing, AMSI memory dump-based deobfuscation, and post-processing of the deobfuscation results. First, the AST-Parser employs a hierarchical traversal to identify all PipelineAST code blocks in the obfuscated script, merging adjacent ASTs of the same type to create complete MergedASTs. Subsequently, during memory dump-based deobfuscation, all MergedASTs are executed in a simulated manner. A specially developed tool is then used to obtain the results from the AMSI dump, extracting the plain code corresponding to the PipelineAST. Concurrently, traverse all subtrees through depth first approach to restore obfuscated content by emulating the execution of strings. This part of our work has been published at https://404nfd.top for accessibility by the security community. Due to space limitations, this paper will not delve into the implementation details of the deobfuscation tool.

3.3. Subtree decomposition and node embedding for multi-type grammar trees

3.3.1. Splitting and extraction of multiple types of AST

PowerShell encompasses over 70 distinct types of ASTs; however, the core syntax and logical functions can be effectively categorized into seven primary types of subtrees, as illustrated in Table 1. Among these, the PipelineAST and AssignmentStatementAST constitute the fundamental code structure of the sample. The PipelineAST represents sequences of one or more commands linked via a pipeline, while the AssignmentStatementAST denotes variable assignment operations. Additionally, the FunctionDefinitionAST outlines the body of a function, and the IfStatementAST, ForeachStatementAST, and WhileStatementAST depict various control flow structures. In scripts designed for repetitive tasks, these ASTs enable developers to create complex and powerful automation.

By analyzing these key AST subtrees, it is possible to effectively identify signatures of both normal behavior and malicious activities

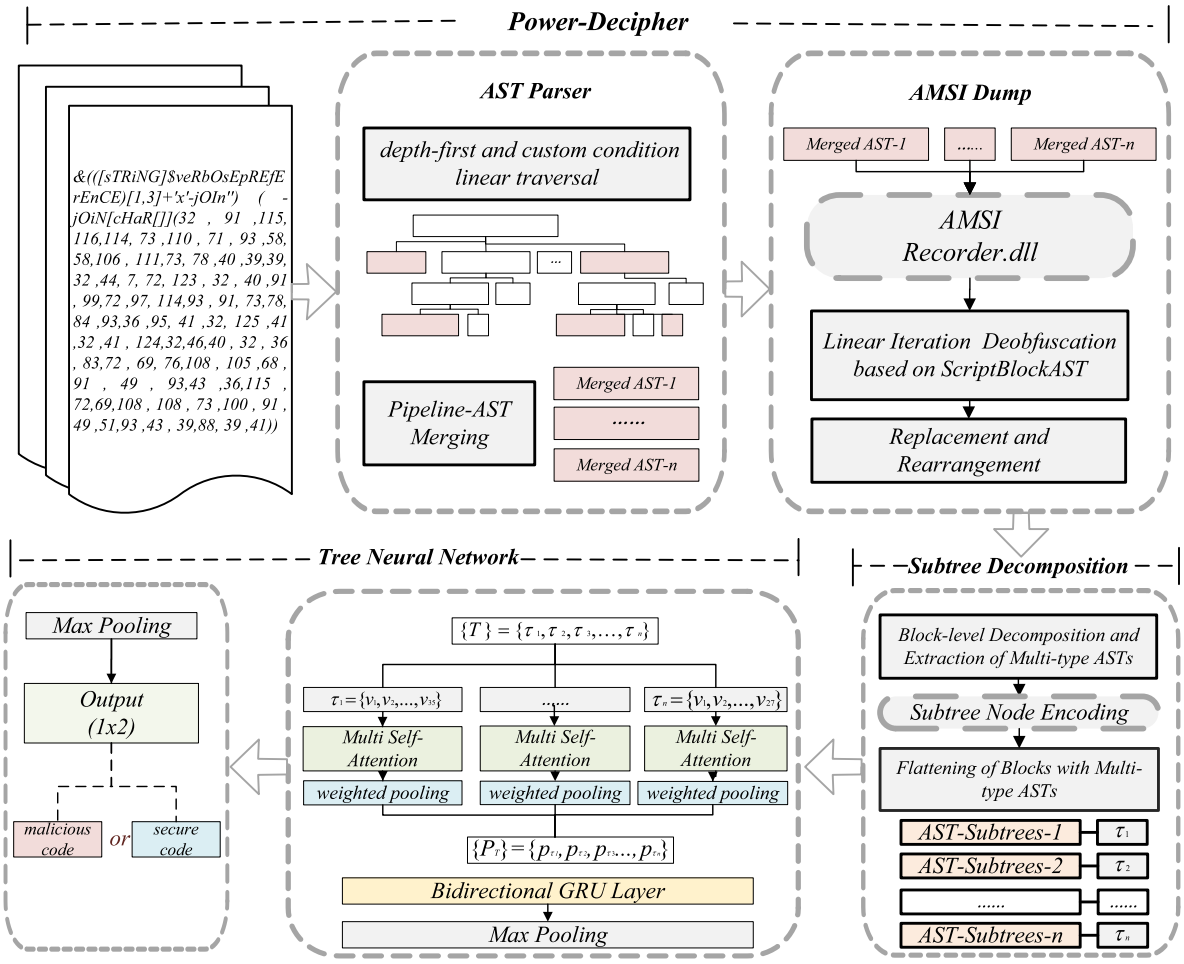

Fig. 1. Power-ASTNN.

within PowerShell code. Collectively, these seven types of subtrees comprise the AST type set $Typeset$ as presented in Eq. (1).

$$Typeset = \{PipelineAST, \ldots, TryStatementAST\} \tag{1}$$

In contrast to ASTNN (Zhang et al., 2019), which recursively collects all subtree nodes, we develop an algorithm to efficiently extract all subtree types within the corresponding $Typeset$ in the script. Fig. 2 illustrates the depth-first traversal process used to decompose the ASTs of the deobfuscated script (S1) and generate the target script file (S2). S1 can be parsed into four major AST types: Add-Type-AssemblyName PresentationCore is transformed into PipelineAST; function definitions are converted to FunctionDefinitionAST; assignment statements are represented as AssignmentStatementAST; and loop statements correspond to WhileStatementAST. Subtrees not included in the $Typeset$ are indicated by an ellipsis, while the parent subtree type is shown in $Typeset$ if the subtree type is not bold.

Following the decomposition of AST subtrees, all subtree nodes matching the $Typeset$ are organized into the set ST. For each subtree node $st_k \in ST$, the starting and ending offsets within the script are recorded using the ordered sets S_{start} and S_{end}. These subtrees are then extracted character by character into the target script file (S2), with the insertion of start and end delimiters "-||->" and "<-||-". This approach ensures that adjacent nested subtrees in S2 are separated by specific delimiters, maintaining the continuity of the subtrees and reflecting the grammatical structure of the original script, which facilitates subsequent feature extraction from the script.

3.3.2. Flattening nested ASTs

To enhance the comprehension of the semantics associated with nodes nested within a subtree, we develop an in-block flattening algorithm, as illustrated in Algorithm 1. This algorithm recursively flattens the nested statement trees of each subtree type. Specifically, it traverses each nested subtree index list: if the current element is a nested subtree, the algorithm continues recursive traversal; if it is a tree node, the element is added to the result list. This recursive process ultimately yields a flattened subtree statement devoid of any nested subtrees.

Algorithm 1 Recursive List Traversal

```
Require: origin_list
Ensure: flatten_list
if isinstance(origin_list, int) then
    flatten_list.append(origin_list)
    return
end if
for item in origin_list do
    if isinstance(item, list) then
        RecursiveListTraversal(item, flatten_list)
    else
        flatten_list.append(item)
    end if
end for
```

3.3.3. Word embedding

Malicious samples typically exhibit differences from benign code in the naming conventions of functions and variables, often employing

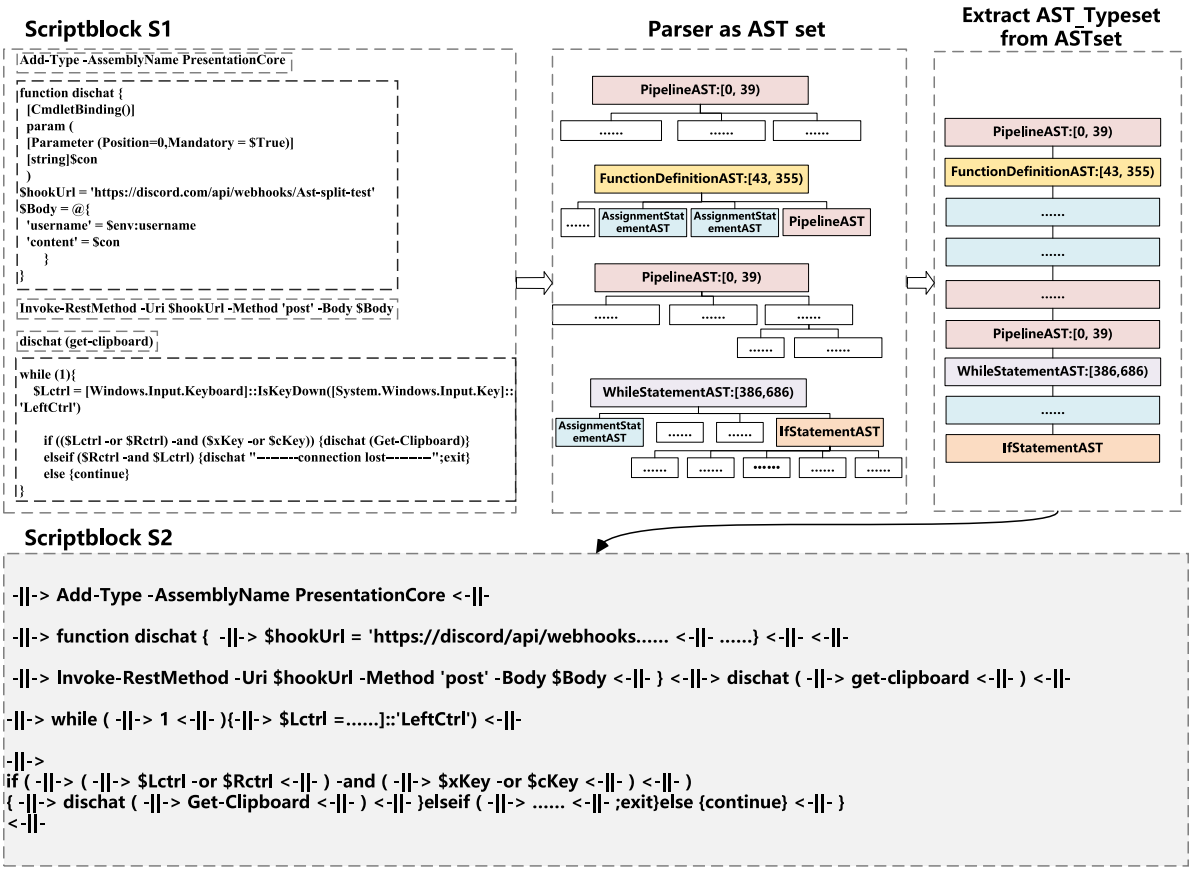

Fig. 2. AST subtree decomposition process.

Table 2
Special Token Processing Methods.

Example of Special Processing	Symbol Example and Processing Method
As a separate token	‘, ‘.’, ‘:’, ‘::’, ‘$’
As a basis for tokenization	Space, newline
Filtered	Single quote, double quote, backtick
Array conversion	Characters matching 0x[0-9a-z][0-9a-z] are replaced with 0 × 11
String conversion	Renamed to stringconstant

unordered names or incorporating special characters. We treat these special characters as tokens for analysis. A summary of the processing of special characters is provided in Table 2.

To train *Word2Vec*, we employ the *Skip-Gram* model with a *Min Count* threshold set to 3, and we conduct selective training for 1000 iterations to obtain the pre-training parameters $W_e \in \mathbb{R}^{|V|\times d}$, where $\mathbb{R}$ denotes the number of rows of the pre-trained embedding parameters, $|V|$ denotes the size of the vocabulary list, and d represents the embedding dimension of each word.

For nodes $n \in \{S1\}$, the embedding is represented as (2):

$$v_n = W_e^T x_n \tag{2}$$

where x_n is the index of the corresponding word, and v_n is the associated d-dimensional vector. Consequently, all nodes within the seven types of subtrees, delineated by delimiters, can be obtained through the word index x_n and the pre-trained parameters.

After deobfuscation, child tree decomposition, flattening and embedding of nodes, the source code of the PowerShell script is converted to a word index representation of a series of continuous, non-overlapping AST subtrees $T = \{\tau_1, \ldots, \tau_n\}$, and the type of each subtree belongs to $Typeset$. Each subtree contains a varying number of nodes.

3.4. Script representation and classification model based on AST

3.4.1. Model structure

Fig. 3 illustrates the overall structure of the model, which comprises two layers: AST node semantic learning and AST sequence semantic learning.

In the AST node semantic learning phase, each subtree is trained independently using a multi-head self-attention mechanism. Each node in the subtree computes its attention score via the Softmax function after being transformed by the *Query* (Q), *Key* (K), and *Value* (V) vectors. Following concatenation and linear transformation, the model derives the attention representations and weights for all nodes, ultimately generating the final representation $PT = \{p_{\tau_1}, \ldots, p_{\tau_n}\}$ of each subtree through weighted pooling.

During the AST sequence semantic learning phase, a bidirectional gated recurrent unit (Bi-GRU) is employed to capture the sequential execution semantics of the subtrees and aggregate their features. The GRU selectively retains or discards information, enabling it to capture long-term dependencies within the sequence data. Its bidirectional architecture allows for the consideration of both preceding and subsequent contextual information. The final output, Y, represents the overall representation of the sample. In a binary classification dataset, after the max pooling layer, the model produces a 1×2 classification result vector.

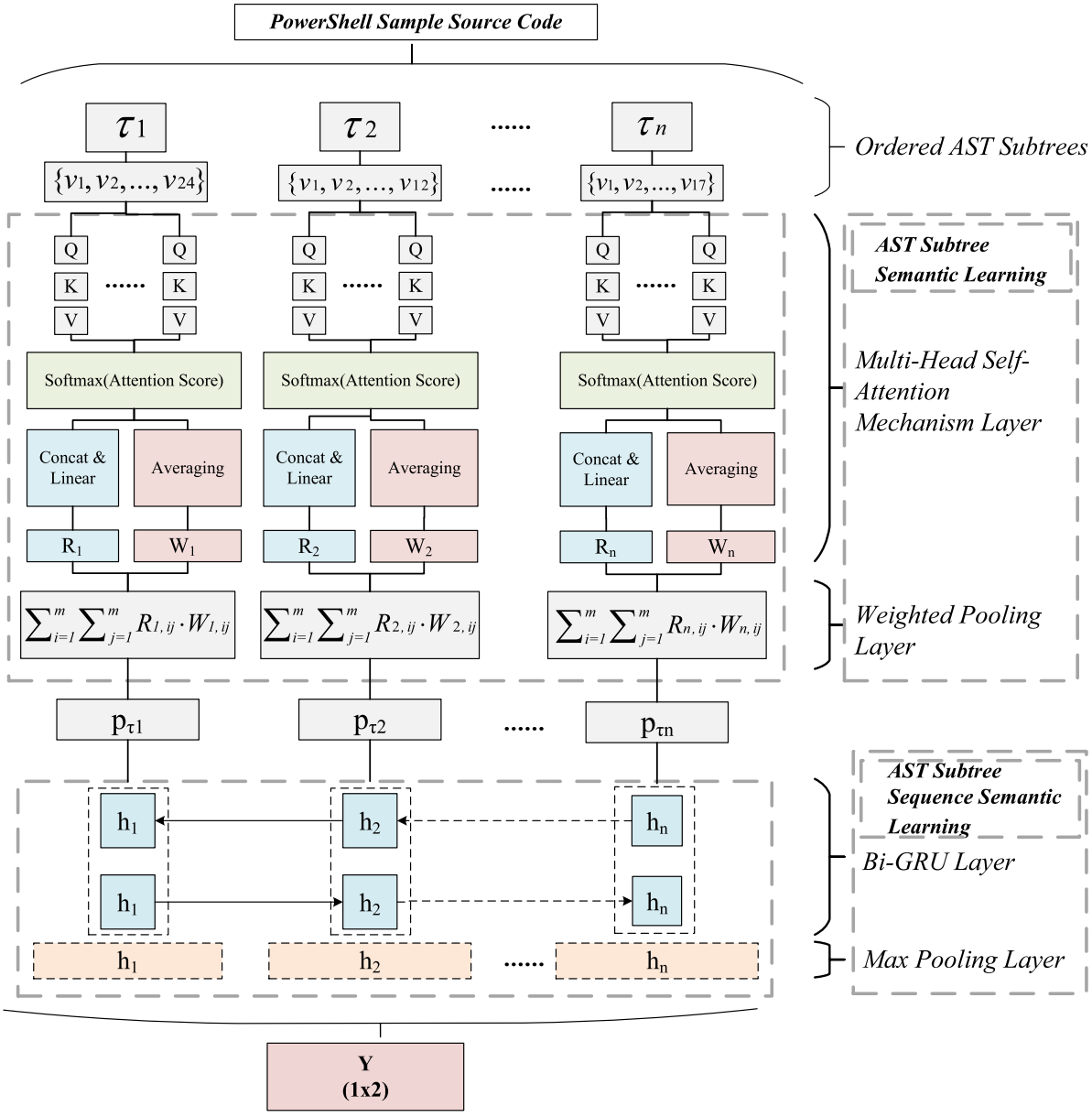

Fig. 3. Model structure detection based on abstract syntax trees.

3.4.2. Semantic learning of AST nodes

In this study, a subtree decomposition of the PowerShell script is conducted to derive the corresponding ordered subtree set $T = \{\tau_1, \ldots, \tau_n\}$. Each subtree τ_i is associated with an ordered node set $\{v_{i1}, v_{i2}, \ldots, v_{im}\}$, leading to the construction of a semantic representation set $PT = \{p_{\tau_1}, \ldots, p_{\tau_n}\}$ for the PowerShell script.

The process of learning the semantic representation for each AST subtree involves two critical steps. First, the semantic relationships among the nodes within the subtree are analyzed to produce a weighted attention representation and a weight matrix for each node. Second, a vector representation for each subtree is constructed through weighted pooling, utilizing the newly obtained node representations and the inter-node weight matrix.

For a given subtree $\tau_k = \{v_{k1}, v_{k2}, \ldots, v_{km}\}$, let h' denote the number of attention heads and $d_{head} = \frac{d}{h'}$ represent the dimension of each attention head. We employ Equation (3) to perform a linear transformation on all nodes, subsequently calculating both the attention weights and attention representations for each node.

$$Q_{ki} = v_{ki}W_{ki}^{Q}, \quad K_{ki} = v_{ki}W_{ki}^{K}, \quad V_{ki} = v_{ki}W_{ki}^{V} \tag{3}$$

Where $Q_{ki}, K_{ki}, V_{ki} \in \mathbb{R}^{1\times d_{head}}$ denote the query, key and value of node v_{ki}, respectively, and $W_{ki}^{Q}, W_{ki}^{K}, W_{ki}^{V} \in \mathbb{R}^{d_{head}\times d_{head}}$ denote the corresponding trainable parameter matrices, which are optimized by backpropagation during training. To facilitate the model's ability to capture relationships and features within the input sequence, the ith node v_{ki} of the subtree τ_k is projected into a higher dimensional space via a linear transformation in (3).

In the current attention head h', the attention score of node v_{ki} relative to all nodes $\{v_{k1}, v_{k2}, \ldots, v_{km}\}$ is calculated by Eq. (4), which calculates the dot product of the key vectors of Q_{ki} and $\{v_{k1}, v_{k2}, \ldots, v_{km}\}$.

$$A\_Score_h(Q_{ki}, K_{k1\to km}) = \left[\frac{Q_{ki}K_{k1}}{\sqrt{d_{head}}}, \ldots, \frac{Q_{ki}K_{km}}{\sqrt{d_{head}}}\right] \tag{4}$$

Eq. (5) transforms these attention scores into a probability distribution, yielding the weight vector for node v_{ki}.

$$A\_Weight_{h,ki} = softmax(A\_Score_h(Q_{ki}, K_{k1\to km})) \tag{5}$$

This vector quantifies the attention each node in subtree τ_k pays to all nodes within the current attention head. Additionally, Eq. (6) calculates the degree of association between nodes v_{ki} and v_{kj}.

$$R_{h,ki} = \sum_{j=1}^{m} A\_Weight_{h,kij} \cdot V_{kj} \tag{6}$$

For $j \subseteq \{1, \ldots, m\}$, there exists a corresponding value vector V_{kj}. The value of $A\_Weight_{h,kij}$ indicates the strength of association between nodes v_{ki} and v_{kj}. The product of these two vectors represents the attention representation of node v_{ki} based on v_{kj}. By summing these representations across all nodes, we obtain the weighted attention representation $R_{h,ki} \in \mathbb{R}^{1\times d_{head}}$ for the current attention head node v_{ki}.

For all m nodes in the subtree τ_k, Eqs. (4), (5), and (6) are utilized to obtain the weighted attention representation matrix $R_{k,h} \in \mathbb{R}^{m\times d_{head}}$ of the m nodes in the current attention.

The h' attention heads correspond to h' subspaces, with each head performing self-attention within its designated subspace to capture distinct node dependencies and patterns. The weight matrix W_O is updated alongside R_k during backpropagation, while d_k is the offset matrix. To obtain the final representation of all nodes $R_k \in \mathbb{R}^{m\times d}$, multi-head concatenation is executed as described in Eq. (7).

$$R_k = [R_{k,h1} \parallel R_{k,h2} \parallel \ldots \parallel R_{k,hh'}]W_o + d_k \tag{7}$$

Similarly, the attention weight representation for all nodes $W_k \in \mathbb{R}^{m\times m}$ is derived by averaging the attention weights $A\_Weight_{h,\text{head},k} \in \mathbb{R}^{m\times m}$, as shown in Eq. (8).

$$W_k = \frac{\sum_{p=1}^{h'} A\_Weight_{hp,k}}{h'} \tag{8}$$

This process elucidates the extent to which each node in subtree τ_k attends to the others, enabling the calculation of the average attention weight.

The weighted pooling operation elegantly consolidates information from multiple nodes within a subtree into a singular representation. This approach effectively captures the essential features and insights of the subtree as a cohesive entity, rather than simply aggregating the information from individual nodes. For a given set $T = \{\tau_1, \ldots, \tau_n\}$, once the multi-head self-attention mechanism has been applied to each subtree, the resulting content is aggregated and dimensionally reduced as described in Eq. (9).

$$\{p_{\tau_1}, \ldots, p_{\tau_n}\} = \tag{9}$$

$$\left\{ \sum_{i=1}^{m}\sum_{j=1}^{m} R_{1,ij} \cdot W_{1,ij}, \ \ldots, \sum_{i=1}^{m}\sum_{j=1}^{m} R_{n,ij} \cdot W_{n,ij} \right\} \tag{10}$$

This step is achieved through weighted pooling across all nodes, resulting in $\{p_{\tau_1}, \ldots, p_{\tau_n}\}$, which captures the key features and information of the subtree as a cohesive entity rather than merely reflecting the characteristics of individual nodes.

3.4.3. Semantic AST sequence learning

To effectively represent the sequential characteristics of the semantic representation set $PT = \{p_{\tau_1}, p_{\tau_2}, \ldots, p_{\tau_n}\}$ corresponding to the script, we employ a Bi-GRU to learn the overall sequential semantics of the script.

To address the gradient vanishing and explosion issues inherent in RNNs, the reset and update gates have been incorporated into the GRU architecture. In this synchronous propagation process, the semantic representation sequence of the input reflects the logical order in which the code is executed. The GRU consists of four core equations that define the update gate (10), reset gate (11), candidate hidden state (12), and final hidden state (13).

$$z_t = \sigma(W_z p_{\tau_t} + U_z h_{t-1} + b_z) \tag{11}$$

$$r_t = \sigma(W_r p_{\tau_t} + U_r h_{t-1} + b_r) \tag{12}$$

$$\tilde{h}_t = \tanh(W_p p_{\tau_t} + U(r_t \odot h_{t-1}) + b_h) \tag{13}$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \tag{14}$$

In this context, z_t in Eq. (10) represents the update gate at time step t, determining the extent to which past information is retained in the current state. The reset gate r_t in Eq. (11) aids the model in discarding irrelevant information for the current task. Both gates utilize weight matrices W_z, U_z, W_r and U_r, along with bias terms b_z and b_r; h_{t-1} denotes the hidden state from the previous time step, and σ is the sigmoid activation function that maps inputs to the interval (0, 1). In Eq. (12), $\tilde{h}_t$ is the candidate hidden state at time step t, calculated based on r_t and z_t. The tanh function is employed to map the internal values to the interval (−1, 1), enabling the capture of both positive and negative features. The final hidden state h_t is derived from the weighted sum of h_{t-1} and the candidate hidden state in Eq. (13).

To further enhance the GRU's ability to capture contextual information from the sentence tree, a bidirectional GRU is incorporated into the final hidden state ht, defined as $ht = [\overrightarrow{ht}, \overleftarrow{ht}], t \in [1, T]$. The candidate hidden state is calculated as $\tilde{h}_t = tanh(W p_{\tau_t} + U(r_t \odot h_{t-1}) + b_h)$, forming a comprehensive representation of the current subtree by concatenating information from both the forward and backward GRUs.

The model output is iteratively trained through three steps: first, the difference between the model output and the actual label is computed using cross-entropy; second, the gradient of the loss function with respect to the network weights is calculated; and third, the model weights are updated using the $Adamax$ optimizer.

4. Experimental results and analysis

4.1. Dataset construction

Our dataset is sourced from five open-source projects: Datacon2022, Malware Bazaar2 (MalwareBazaar, 2020), VirusTotal Academic Samples (MalwareBazaar, 2023), Fang Yong (Fang et al., 2021), and the GitHub open-source PS1 sample repository. All samples have been deduplicated, organized, and converted to UTF-8 format. Ultimately, we construct a comprehensive dataset, Dataset-C, comprising 5606 benign samples and 5338 malicious samples. Additionally, to evaluate the impact of obfuscation on the detection model, we create an obfuscation dataset, Dataset-O, containing 1517 benign obfuscations and 1583 malicious obfuscations based on Dataset-C.

4.2. Experimental setup and evaluation metrics

All experiments are conducted on a workstation equipped with an Intel Core i7-11800H CPU, a GTX 4090 GPU, and 32 GB of RAM. We compare Power-ASTNN with baseline models (Hendler (Hendler et al., 2020), Mimura (Mimura and Tajiri, 2021), MPSAutodetect (Alahmadi et al., 2022)) using both the comprehensive and obfuscated datasets.

The evaluation metrics for the model's detection performance include: Accuracy, defined as the ratio of correctly classified samples to the total number of samples; Precision, the proportion of actual positive samples among those classified as positive; Recall, the ratio of correctly identified malicious samples to all malicious samples; and F1-Score, the harmonic mean of precision and recall. Additionally, we introduce the Area Under the Curve (AUC), which measures the area under the ROC curve with a value range from 0 to 1. The AUC index comprehensively considers true positive rate (TPR) and false positive rate (FPR) across all classification thresholds, thus mitigating the effects of imbalanced data.

Table 3
Comparison of the detection performance on the mixed dataset.

Dataset	Method	Accuracy	Precision	Recall	F1-Score	AUC
Dataset-C	Hendler (Hendler et al., 2020)	84.96	79.66	95.32	86.79	90.27
	Mimura (Mimura and Tajiri, 2021)	97.22	97.88	96.27	97.07	99.51
	MPSAutodetect (Alahmadi et al., 2022)	93.19	93.07	93.53	93.45	94.38
	Power-ASTNN	**98.87**	**98.63**	**99.20**	**98.91**	**99.57**

Table 4
Comparison of the detection performance on the obfuscated dataset.

Dataset	Method	Accuracy	Precision	Recall	F1-Score	AUC
Dataset-O	Hendler (Hendler et al., 2020)	84.42	80.25	92.77	86.06	87.07
	Mimura (Mimura and Tajiri, 2021)	86.30	94.29	80.49	86.84	96.19
	MPSAutodetect (Alahmadi et al., 2022)	75.51	74.25	75.87	74.67	68.90
	Power-ASTNN	**97.45**	**98.75**	**97.90**	**99.80**	**98.35**

Table 5
Comparison of detection performance under different granularity.

Granularity	Detection Method	Accuracy	Precision	Recall	F1-Score	AUC
Tokens	Multi-Head Attention + Bi-GRU	81.89%	85.85%	77.07%	81.22%	90.79%
	Multi-Head Attention + Bi-LSTM	83.21%	85.21%	81.01%	83.06%	91.10%
Lines	Multi-Head Attention + Bi-GRU	93.70%	95.39%	91.05%	95.56%	91.79%
	Multi-Head Attention + Bi-LSTM	94.35%	95.20%	93.10%	95.70%	95.96%
ASTs	**Multi-Head Attention + Bi-GRU**	**98.87%**	**98.63%**	**99.20%**	**98.91%**	**99.57%**
	Multi-Head Attention + Bi-LSTM	97.33%	99.05%	95.75%	97.37%	97.32%

In the experiments, the dimension of the tokens for training the Word2Vec model is set to 128. The number of heads in the multi-head self-attention mechanism of Power-ASTNN is configured to 8. The Bi-GRU comprise one recurrent layer with a network layer dimension of 100. The training loop is executed for 15 epochs, with a batch size of 32.

4.3. Comparative experiment

Table 3 presents the testing results for the four models on the comprehensive dataset. Power-ASTNN demonstrates superior detection capabilities, achieving recall and F1-Score improvements of 2.93% and 2.84%, respectively, over the current best-performing scheme, with both recall and AUC values exceeding 99%. This indicates the model's effectiveness in minimizing false positives for benign samples and reducing missed detections of malicious samples.

Table 4 displays the results of the experiment on the obfuscated dataset. When solely using obfuscated samples, the detection performance of Power-ASTNN exhibits the least decline. In contrast, the detection performance of other models is adversely affected to varying extents, with MPSAutodetect experiencing the most significant drop. This decline is attributed to the conversion all samples into ASCII code representations during the design phase, which hindered the model's training effectiveness, resulting in an AUC value below 70%.

The detection performance of the Hendler model on the obfuscated dataset is relatively unaffected. This can be attributed to the fact that both datasets utilize complete AMSI memory dump results, which include a significant amount of intermediate execution code. The absence of post-processing to eliminate this intermediate code hampers further improvement in the model's detection results.

For detecting obfuscated samples, the distinct construction methods of malicious and benign obfuscations allow Power-ASTNN's fine-grained semantic learning to effectively address this adversarial detection task. While the Mimura model exhibits a low false negative rate for malicious samples in the obfuscated dataset, its overall performance remains inferior to that of Power-ASTNN. The MPSAutodetect model struggles to effectively handle obfuscated samples, and although the Hendler model demonstrates relatively stable performance on the confused dataset, its accuracy remains low.

4.4. Adversarial attacks resisting experiment

Fig. 4 illustrates the detection results of four models – Hendler, Mimura, MPSAutodetect, and Power-ASTNN – across a range of adversarial sample proportions from 0% to 100%. Power-ASTNN demonstrates high robustness in adversarial scenarios, exhibiting less decline in detection performance compared to Hendler and MPSAutodetect as the proportion of adversarial samples increases. Although MPSAutodetect employs a three-layer stacked autoencoder with noise reduction processing, its detection performance sharply deteriorates when the proportion of adversarial samples rises from 40% to 60%. Mimura utilizes a word frequency-based vector construction method, which can yield better detection results when malicious samples in the dataset exhibit high similarity in word frequency. However, this approach lacks the generalization ability to identify malicious behaviors with significant structural differences, resulting in underperformance when facing new or unknown samples.

Additionally, we observe that processing incomplete sample inputs likely contributes to the degradation of detection capabilities in adversarial scenarios. The Hendler model accepts only the first 1000 characters and nodes, while MPSAutodetect accepts only the first 1000 ASCII codes. Consequently, if malicious code is located in the latter part of the sample, the models may fail to detect and classify it correctly. When the proportion of adversarial samples exceeds 40%, both the Hendler and MPSAutodetect models experience a sharp decline in detection performance, with MPSAutodetect nearly losing its detection and classification capabilities at 100% adversarial samples.

4.5. Ablation experiment

4.5.1. Contribution of AST decomposer and Bi-GRU

As shown in Table 5, different AST decomposition strategies significantly impact model performance. With Token-level granularity

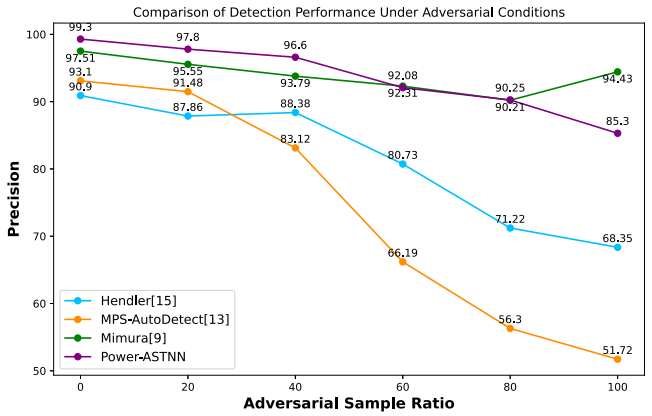

Fig. 4. Comparison of detection performance under adversarial conditions.

Table 6
The Impact of Different Numbers of Self-Attention Heads on AST Node Semantic Learning Performance.

Number of Self-Attention Heads	Accuracy	Precision	Recall	F1-Score	AUC
1	98.46%	98.29%	98.74%	98.47%	98.35%
2	98.46%	98.51%	98.52%	98.51%	98.31%
4	98.76%	98.41%	99.20%	98.80%	98.93%
8	**98.87%**	**98.63%**	**99.20%**	**98.91%**	**99.57%**
12	98.65%	98.23%	99.20%	98.71%	98.94%
16	98.58%	98.07%	99.20%	98.63%	98.86%

Table 7
The Impact of Different Types of GNN on Detection Performance.

Neural Network Type	Accuracy	Precision	Recall	F1-Score	AUC
ASTNN (Zhang et al., 2019)	96.68%	95.43%	98.28%	96.83%	98.10%
Text-CNN (Du et al., 2023)	98.12%	97.36%	95.13%	96.23%	97.12%
Tree-LSTM (Yu et al., 2020)	94.42%	91.32%	90.50%	90.91%	93.59%
Power-ASTNN-8	**98.87%**	**98.63%**	**99.20%**	**98.91%**	**99.57%**

decomposition, Bi-LSTM performs slightly better than Bi-GRU, but both exhibit relatively low performance, particularly in Recall and F1-Score. In contrast, when using Lines-level granularity decomposition, both Bi-GRU and Bi-LSTM show improved and comparable performance, indicating that line-by-line decomposition effectively enhances model efficacy. Ultimately, with AST-level granularity decomposition, both bidirectional GRU and bidirectional LSTM achieve the highest model performance, underscoring the importance of subtree structure decomposition for malicious detection. This approach retains more syntactic structure features compared to direct Token-level or Line-level decomposition.

In summary, the granularity of AST decomposition and the choice of GRU cell type significantly influence the overall detection performance of Power-ASTNN. A well-considered subtree decomposition strategy and appropriate cell selection can substantially enhance Power-ASTNN's performance in PowerShell malicious detection, providing crucial guidance for future model optimization and customization.

4.5.2. The influence of the number of self-attention heads

After identifying the optimal subtree decomposition strategy, we analyze the impact of the number of heads in the multi-head attention mechanism on AST node semantic learning performance. The results, presented in Table 6, indicate that the multi-head self-attention mechanism consistently yields the best results across nearly all metrics. When the number of attention heads is four or greater, the miss rate for malicious samples in the Power-ASTNN model stabilizes at approximately 99.20. Setting the number of heads to 8 maximizes the effectiveness of the multi-head self-attention mechanism in learning AST node semantics, achieving an F1-Score of 98.91 and an AUC value of 99.57. However, as the number of heads increases beyond this point, model accuracy, Precision, F1-Score, and AUC values decline.

4.5.3. The impact of various neural network learning methods

We also compare the learning capabilities of different neural networks applied in Power-ASTNN, including: (1) the basic ASTNN (Zhang et al., 2019), which leverages AST structure to learn syntactic features of code; (2) Text-CNN (Du et al., 2023), which treats the code text of a subtree as one-dimensional images, using a convolutional neural network to capture associations between adjacent words; and (3) Tree-LSTM (Yu et al., 2020), which employs a Bi-LSTM to calculate the representation of parent nodes by integrating information from child nodes. The comparison results are shown in Table 7, where the Text-CNN model utilizes two convolutional layers with 64 and 128 convolutional kernels, respectively, to derive vector representations for each subtree through a fully connected layer.

Notably, Text-CNN outperforms Bi-LSTM across all metrics, which can be attributed to its superior ability to capture local contextual information. While Bi-LSTM is effective at understanding long-distance dependencies, it is less efficient than convolutional neural networks in processing long sequences typical of PowerShell scripts. In contrast, Power-ASTNN employs a tree neural network and a multi-head self-attention mechanism to accurately capture node and sequence features, resulting in enhanced detection performance.

4.5.4. Time efficiency analysis

We further evaluate the performance of Power-ASTNN in terms of sample pre-processing efficiency and detection efficiency. This evaluation included a comparison with Power-RCNN, a self developed lightweight detection method utilizing LSTM and convolutional neural networks for syntax tree sequence extraction and subtree semantic learning. The pre-processing time for Power-RCNN and Power-ASTNN are 227.5 ms and 2500.7 ms, respectively, while the single sample detection times are 207.5 ms and 194.9 ms, respectively. These results indicate that Power-ASTNN incurs higher processing times due to the complex AST analysis and subtree decomposition involved.

5. Conclusion

This paper presents and a fine-grained detection model based on abstract syntax trees for malicious PowerShell scripts detection which is build on a self developed deobfuscation tool. The model's effectiveness is primarily attributed to its carefully designed AST node decomposition strategy and its efficient semantic learning capabilities. Notably, optimal detection performance is achieved when the number of self-attention heads is set to 8. Experimental results demonstrate that Power-ASTNN exhibits high robustness in detecting obfuscated data samples and in adversarial scenarios. Overall, our model is well-suited for complex and generalized detection tasks. The relevant software has been released in open source community for use by security researchers. Future work will focus on improving pre-processing efficiency and enhancing robustness when addressing complex samples.

CRediT authorship contribution statement

Sanfeng Zhang: Writing – review & editing, Writing – original draft, Supervision, Resources, Project administration, Methodology, Conceptualization. **Shangze Li:** Software, Methodology, Data curation, Conceptualization. **Juncheng Lu:** Writing – review & editing, Visualization, Software, Methodology, Data curation. **Wang Yang:** Writing – review & editing, Supervision, Funding acquisition.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Wang Yang reports financial support was provided by National key R&D program of China. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We would like to thank the anonymous reviewers for their careful reading and insightful suggestions. This research has been partially funded by the National Key R&D Program under Grants no. 2022YFB3104601.

Data availability

Data will be made available on request.

References

Al-Saleh, Mohammed Ibrahim, Espinoza, Antonio M., Crandall, Jedediah R., 2013. Antivirus performance characterisation: system-wide view. IET Inf. Secur. 7 (2), 126–133.

Alahmadi, Amal, Alkhraan, Norah, BinSaeedan, Wojdan, 2022. MPSAutodetect: a malicious powershell script detection model based on stacked denoising auto-encoder. Comput. Secur. 116, 102658.

Alon, Uri, Zilberstein, Meital, Levy, Omer, Yahav, Eran, 2018. A general path-based representation for predicting program properties. ACM SIGPLAN Not. 53 (4), 404–419.

Anon, 2014. 2023 Global threat report. Website. https://go.crowdstrike.com/2023-global-threat-report.

Anon, 2015. Windows matrix. Website. https://attack.mitre.org/matrices/enterprise/windows/.

Barr-Smith, Frederick, Ugarte-Pedrero, Xabier, Graziano, Mariano, Spolaor, Riccardo, Martinovic, Ivan, 2021. Survivalism: Systematic analysis of windows malware living-off-the-land. In: 2021 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 1557–1574.

Bode, Alexander, Warnars, Niels, Velasco, Leandro, de Novais Marques, Joao, 2020. Detecting fileless malicious behaviour of .NET C2 agents using ETW.

Brengel, Michael, Rossow, Christian, 2021. {YARIX}: Scalable {yara-based} malware intelligence. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 3541–3558.

Chai, Huajun, Ying, Lingyun, Duan, Haixin, Zha, Daren, 2022. 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, IEEE, pp. 295–306.

Cheng, Binlin, Leal, Erika A., Zhang, Haotian, Ming, Jiang, 2023. On the feasibility of malware unpacking via hardware-assisted loop profiling. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 7481–7498.

Cheng, Binlin, Ming, Jiang, Leal, Erika A, Zhang, Haotian, Fu, Jianming, Peng, Guojun, Marion, Jean-Yves, 2021. {Obfuscation-resilient} executable payload extraction from packed malware. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 3451–3468.

Choi, Sunoh, 2020. Malicious powershell detection using attention against adversarial attacks. Electronics 9 (11), 1817.

Deng, Zhongyang, Xu, Ling, Liu, Chao, Yan, Meng, Xu, Zhou, Lei, Yan, 2022. Fine-grained co-attentive representation learning for semantic code search. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 396–407.

Du, Gewangzi, Chen, Liwei, Wu, Tongshuai, Zheng, Xiong, Cui, Ningning, Shi, Gang, 2023. Cross domain on snippets: BiLSTM-TextCNN based vulnerability detection with domain adaptation. In: 2023 26th International Conference on Computer Supported Cooperative Work in Design. CSCWD, IEEE, pp. 1896–1901.

Fang, Yong, Zhou, Xiangyu, Huang, Cheng, 2021. Effective method for detecting malicious PowerShell scripts based on hybrid features. Neurocomputing 448, 30–39.

Fass, Aurore, Krawczyk, Robert P, Backes, Michael, Stock, Ben, 2018. Jast: Fully syntactic detection of malicious (obfuscated) javascript. In: Detection of Intrusions and Malware, and Vulnerability Assessment: 15th International Conference, DIMVA 2018, Saclay, France, June 28-29, 2018, Proceedings 15. Springer, pp. 303–325.

Gaber, Matthew G., Ahmed, Mohiuddin, Janicke, Helge, 2024. Malware detection with artificial intelligence: A systematic literature review. ACM Comput. Surv. 56 (6), 1–33.

Galloro, Nicola, Polino, Mario, Carminati, Michele, Continella, Andrea, Zanero, Stefano, 2022. A systematical and longitudinal study of evasive behaviors in windows malware. Comput. Secur. 113, 102550.

Gopinath, Mohana, Sethuraman, Sibi Chakkaravarthy, 2023. A comprehensive survey on deep learning based malware detection techniques. Comput. Sci. Rev. 47, 100529.

Hendler, Danny, Kels, Shay, Rubin, Amir, 2018. Detecting malicious powershell commands using deep neural networks. In: Proceedings of the 2018 on Asia Conference on Computer and Communications Security. pp. 187–197.

Hendler, Danny, Kels, Shay, Rubin, Amir, 2020. Amsi-based detection of malicious powershell code using contextual embeddings. In: Proceedings of the 15th ACM Asia Conference on Computer and Communications Security. pp. 679–693.

Hou, Yiwei, Guo, Lihua, Zhou, Chijin, Xu, Yiwen, Yin, Zijing, Li, Shanshan, Sun, Chengnian, Jiang, Yu, 2024. An empirical study of data disruption by ransomware attacks. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. pp. 1–12.

Kara, Ilker, 2023. Fileless malware threats: Recent advances, analysis approach through memory forensics and research challenges. Expert Syst. Appl. 214, 119133.

Li, Zhenyuan, Chen, Qi Alfred, Xiong, Chunlin, Chen, Yan, Zhu, Tiantian, Yang, Hai, 2019. Effective and light-weight deobfuscation and semantic-aware attack detection for powershell scripts. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 1831–1847.

Li, Shijia, Jia, Chunfu, Qiu, Pengda, Chen, Qiyuan, Ming, Jiang, Gao, Debin, 2022. Chosen-instruction attack against commercial code virtualization obfuscators. In: Proceedings of the 29th Network and Distributed System Security Symposium.

Lin, Chen, Ouyang, Zhichao, Zhuang, Junqing, Chen, Jianqiang, Li, Hui, Wu, Rongxin, 2021. Improving code summarization with block-wise abstract syntax tree splitting. In: 2021 IEEE/ACM 29th International Conference on Program Comprehension. ICPC, IEEE, pp. 184–195.

Liu, Chao, Xia, Bin, Yu, Min, Liu, Yunzheng, 2018. PSDEM: a feasible de-obfuscation method for malicious PowerShell detection. In: 2018 IEEE Symposium on Computers and Communications. ISCC, IEEE, pp. 825–831.

Luo, Chenke, Ming, Jiang, Fu, Jianming, Peng, Guojun, Li, Zhetao, 2023. Reverse engineering of obfuscated lua bytecode via interpreter semantics testing. IEEE Trans. Inf. Forensics Secur. 18, 3891–3905.

Malandrone, Giuseppe Mario, Virdis, Giovanni, Giacinto, Giorgio, Maiorca, Davide, et al., 2021. Powerdecode: a powershell script decoder dedicated to malware analysis. In: Proceedings of the Italian Conference on Cybersecurity, ITASEC 2021, vol. 2940, CEUR-WS. org, pp. 219–232.

MalwareBazaar, 2020. Introducing MalwareBazaar. Website. https://abuse.ch/blog/introducing-malwarebazaar/.

MalwareBazaar, 2023. VirusTotal. https://www.virustotal.com/gui/home/upload.

Mimura, Mamoru, Tajiri, Yui, 2021. Static detection of malicious PowerShell based on word embeddings. Internet Things 15, 100404.

Palo Alto Networks, 2010a. 2023 unit 42 ransomware and extortion report. Website. https://start.paloaltonetworks.com/2023-unit42-ransomware-extortion-report/.

Palo Alto Networks, 2010b. New tool set found used against organizations in the middle east, Africa and the US. Website. https://unit42.paloaltonetworks.com/new-toolset-targets-middle-east-africa-usa/.

R3MRUM, 2020. R3MRUM/PSDecode. Website. https://github.com/R3MRUM/PSDecode.

Rose, Anthony J., Kabban, Christine M. Schubert, Graham, Scott R., Henry, Wayne C., Rondeau, Christopher M., 2024. Malware classification through Abstract Syntax Trees and L-moments. Comput. Secur. 104082.

Rusak, Gili, Al-Dujaili, Abdullah, O'Reilly, Una-May, 2018. Ast-based deep learning for detecting malicious powershell. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 2276–2278.

Shi, Ensheng, Wang, Yanlin, Du, Lun, Zhang, Hongyu, Han, Shi, Zhang, Dongmei, Sun, Hongbin, 2023. Cocoast: representing source code via hierarchical splitting and reconstruction of abstract syntax trees. Empir. Softw. Eng. 28 (6), 135.

Song, Jihyeon, Kim, Jungtae, Choi, Sunoh, Kim, Jonghyun, Kim, Ikkyun, 2021. Evaluations of AI-based malicious PowerShell detection with feature optimizations. ETRI J. 43 (3), 549–560.

Sun, Weisong, Fang, Chunrong, Miao, Yun, You, Yudu, Yuan, Mengzhe, Chen, Yuchen, Zhang, Quanjun, Guo, An, Chen, Xiang, Liu, Yang, et al., 2023. Abstract syntax tree for programming language understanding and representation: How far are we?. arXiv preprint arXiv:2312.00413.

Tao, Chuanqi, Lin, Kai, Huang, Zhiqiu, Sun, Xiaobing, 2022. Cram: Code recommendation with programming context based on self-attention mechanism. IEEE Trans. Reliab. 72 (1), 302–316.

Ugarte, Denis, Maiorca, Davide, Cara, Fabrizio, Giacinto, Giorgio, 2019. PowerDrive: accurate de-obfuscation and analysis of PowerShell malware. In: Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16. Springer, pp. 240–259.

Xiong, Chunlin, Li, Zhenyuan, Chen, Yan, Zhu, Tiantian, Wang, Jian, Yang, Hai, Ruan, Wei, 2022. Generic, efficient, and effective deobfuscation and semantic-aware attack detection for PowerShell scripts. Front. Inf. Technol. Electron. Eng. 23 (3), 361–381.

Yang, Xiuzhang, Peng, Guojun, Zhang, Dongni, Gao, Yuhang, Li, Chenguang, 2023. PowerDetector: Malicious PowerShell script family classification based on multi-modal semantic fusion and deep learning. China Commun. 20 (11), 202–224.

Yu, Xiang, Li, Guoliang, Chai, Chengliang, Tang, Nan, 2020. Reinforcement learning with tree-lstm for join order selection. In: 2020 IEEE 36th International Conference on Data Engineering. ICDE, IEEE, pp. 1297–1308.

Zhang, Jian, Wang, Xu, Zhang, Hongyu, Sun, Hailong, Wang, Kaixuan, Liu, Xudong, 2019. A novel neural source code representation based on abstract syntax tree. In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, IEEE, pp. 783–794.

Sanfeng Zhang (Member, IEEE) received his Ph.D. degree in computer science from Southeast University, Nanjing, China, in 2008. He was a Visiting Scholar with the University of California, Davis, USA from 2016 to 2017. In 2008, he joined Southeast University, Nanjing, China, where he is currently an associate Professor. His research interests include cyber security and graph neural networks.

Shangze Li received the B.E. degree in information security from College of Computer Science and Engineering, Tianjin University of Science and Technology, Tianjin, China, in 2021. He is currently pursuing the M.E. degree with the School of Cyber Science and Engineering in Southeast University. His main research directions are intrusion detection, and malware detection.

Juncheng Lu is currently pursuing a bachelor's degree in School of Cyber Science and Engineering in Southeast University. His research interests include graph neural networks and malware detection. The focus of malware detection encompasses various languages, including but not limited to PowerShell and Java.

Wang Yang (Member, IEEE) received his Ph.D. degree in computer science from Southeast University, Nanjing, China, in 2009. He was a Visiting Scholar with the University of Pittsburgh, PA, USA from 2014 to 2015. In 2009, he joined Southeast University, Nanjing, China, where he is currently an associate Professor. His research interests include threat intelligence and deep learning.