electronics

MDPI

Article

AST-DF: A New Webshell Detection Method Based on Abstract Syntax Tree and Deep Forest

Chengfeng Dong [1,2] and Daofeng Li [1,2,*]

[1] School of Computer and Electronic Information, Guangxi University, Nanning 530004, China; 2113391007@st.gxu.edu.cn
[2] Guangxi Colleges and Universities Key Laboratory of Multimedia Communications and Information Processing, Guangxi University, Nanning 530004, China
* Correspondence: ldf-0123@gxu.edu.cn

Abstract: Webshell is a kind of web-language-based website backdoor, which is usually used by attackers to control web servers. Due to its dangerous nature, how to detect Webshell effectively has become a hot research topic in current Web security research. With the rapid development of Webshell evasion technology, the existing Webshell detection methods have the problem of insufficient ability to detect unknown Webshells. In order to solve the above problems and achieve effective Webshell detection, this study proposes a Webshell detection method based on the abstract syntax tree (AST) and deep forest (DF) model called AST-DF. AST-DF first extracts the abstract syntax tree from the PHP code; then, the abstract syntax tree sequence is feature extracted and vectorized using N-gram and TF-IDF. Finally, the vectors are imported into the deep forest model for classification to determine whether the PHP code to be detected is a Webshell or not. The experimental results show that AST-DF achieves remarkable effects in the task of detecting PHP-type Webshells, with a 99.61% accuracy rate, and the values of precision, recall, and F1 score are more than 99%.

Keywords: Webshell detection; abstract syntax tree; TF-IDF; deep forest

Citation: Dong, C.; Li, D. AST-DF: A New Webshell Detection Method Based on Abstract Syntax Tree and Deep Forest. *Electronics* **2024**, *13*, 1482. https://doi.org/10.3390/electronics13081482

Academic Editor: Fabio Grandi

Received: 28 February 2024
Revised: 7 April 2024
Accepted: 11 April 2024
Published: 13 April 2024

1. Introduction

With the continuous development of Web technology, the functions of Web applications have gradually become richer and more diversified [1], and this rich functionality provides attackers with a broader attack surface, making Web applications more vulnerable to various types of attacks. Webshell, as a backdoor tool based on Web language, is an important weapon often used by attackers in Web attacks [2]. Attackers often take advantage of vulnerabilities, such as SQL injection [3], arbitrary file upload [4], remote code execution [5], and file inclusion [6], to drop Webshell onto the target system to gain control of the target system server. Meanwhile, attackers can also use Webshell as a springboard to build proxy tunnels to further invade the internal network for deep attacks. In response to the imperative of mitigating Webshell attacks and fortifying network security, extensive research endeavors have been undertaken in the realm of Webshell detection. Notably, investigations have explored the application of machine learning and deep learning techniques to classify packets generated during Webshell communication, thereby advancing Webshell detection through the lens of communication traffic analysis [7–9]. Simultaneously, researchers have directed their attention towards the scrutiny of Webshell files, embarking on the extraction of static or abstract features [10] inherent in these files. Subsequently, machine learning or neural network methodologies are employed for the discernment of Webshells based on these extracted features [11–14]. However, a critical challenge in the Webshell detection landscape arises during the stage of Webshell placement. To avoid being detected by Webshell detection tools, attackers skillfully employ a range of evasion techniques, including encryption, obfuscation, and string splitting, to alter Webshells. These clever manipulations make Webshell appear as normal files. For instance, Figure 1 illustrates how

a traditional PHP one-liner Trojan is obfuscated to remove its original Webshell signature. The development and widespread use of Webshell evasion techniques have made existing Webshell detection methods ineffective at identifying Webshells. Meanwhile, according to W3Techs, PHP is currently the most popular server-side programming language, with a share of 76.4% [15]. Due to the wide use of PHP language, the number of PHP-type Webshells is also considerable. Detection against PHP-type webshells is essential to protect network security.

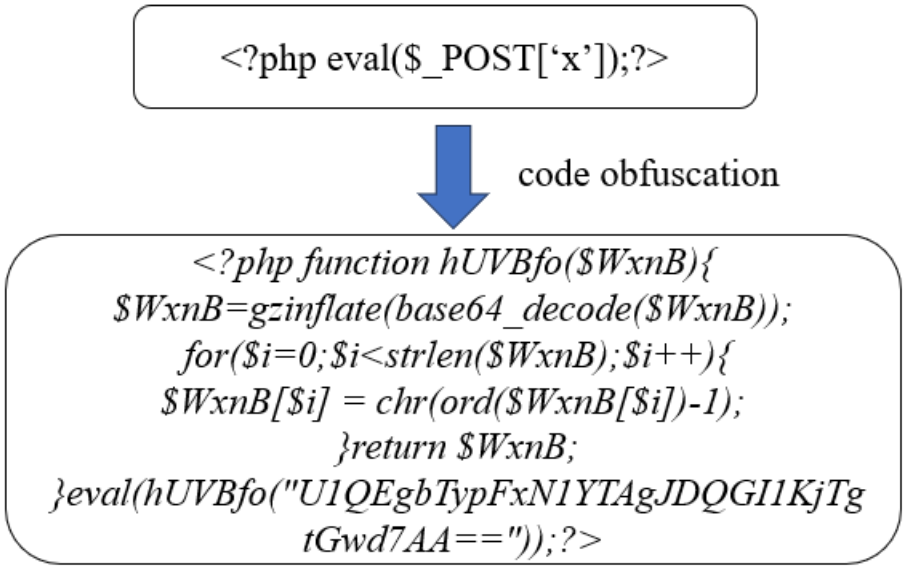

Figure 1. Examples of code obfuscation.

In order to effectively solve the problem that Webshell evasion technology leads to low accuracy for existing Webshell detection methods and poor ability to detect unknown Webshells, this study proposes a new Webshell detection method AST-DF based on the abstract syntax tree and deep forest model, which first extracts the abstract syntax tree (AST) sequence features of the PHP code and then performs the vectorization and feature extraction with N-Gram and TF-IDF, finally detecting the Webshells of the PHP language type by using the deep forest model.

The main contributions of this study are as follows:

1. Propose a PHP-type Webshell detection method named AST-DF, which is based on abstract syntax tree and deep forest model.
2. AST-DF demonstrates effectiveness in detecting PHP-type Webshells, surpassing 99% in accuracy, precision, recall, and F1 score.
3. AST-DF has detection capabilities for unknown PHP-type Webshells due to its use of abstract syntax trees as features.

2. Related Word

Currently, Webshell detection research can be divided into dynamic detection and static detection according to the detection method [16]. Dynamic detection can detect Webshell from both host and network levels, while static detection mainly uses various features of the source code to identify Webshell files and normal files.

In dynamic detection methods, host-based Webshell detection typically employs Hook techniques [17] to monitor the behavior of each file on the server, determining whether it is a Webshell. This approach requires continuous monitoring of the behavior of every file on the server, resulting in a significant drain on system resources. On the other hand, network-based Webshell detection utilizes the content of HTTP data packets generated during communication as features. Machine learning algorithms or deep learning techniques are then employed for classification to achieve Webshell detection. In terms of traditional

machine learning algorithms, SVM algorithms are used to classify HTTP packets for the purpose of Webshell detection, and this method achieves high accuracy and recall rates [8]. And, in terms of deep learning techniques, Zhang et al. combined CNN and LSTM models to classify HTTP packets for the purpose of detecting Webshell and achieved good performance [18]. Le et al. used DNN and rule-based Webshell detection techniques to detect Webshells [19]. While dynamic Webshell detection exhibits good performance in terms of detection effectiveness and the ability to identify unknown Webshells, it is hindered by challenges such as high server resource consumption and deployment difficulties in practical scenarios.

Static detection primarily analyzes the static features of the code, employing machine learning or deep learning algorithms to detect Webshells. In the application of traditional machine learning algorithms for detection, researchers commonly use Bayesian algorithms [20], matrix decomposition algorithms [21], and similar approaches. Since a singular algorithm may not always effectively detect Webshells, some researchers conduct experiments with multiple algorithms, selecting the most effective one for detection [22]. Additionally, there are studies employing ensemble learning algorithms for Webshell detection. For instance, Fang et al. used the random forest algorithm to detect PHP-type Webshells and obtained an accuracy rate of 99.23% [23], Tianmin et al. utilized the XGBoost algorithm [24] for detecting PHP-type Webshells [25], and Cui et al. employed the random forest gradient-boosting decision tree (RF-GBDT) for Webshell detection [26]. Traditional machine learning-based Webshell detection methods offer advantages such as fast detection speed and low system resource consumption. However, they are associated with drawbacks, including lower detection accuracy and limited ability to identify unknown Webshells.

As neural networks have achieved remarkable results in the field of natural language processing, and the nature of the Webshell detection task is a special text binary classification task, there are many researchers applying neural networks to Webshell detection. For instance, Zhou et al. employed Long Short-Term Memory networks for Webshell detection [27]; Lv et al. utilized convolutional neural networks [11]; and Wu et al. leveraged reinforcement learning to enhance CNNs in Webshell detection [13]. Liu et al. achieved Webshell detection across multiple languages through the use of Bidirectional Gated Recurrent Unit networks and attention mechanisms [28]. Neural networks have proven highly effective in Webshell detection, addressing the limitations of traditional machine learning in terms of accuracy and the ability to detect unknown Webshells. However, the resource-intensive nature of neural network training and the requirement for abundant labeled samples pose challenges. In the field of Webshell detection, the scarcity of labeled samples may lead to issues of inadequate generalization when neural networks are trained on limited datasets.

In summary, dynamic Webshell detection methods offer high accuracy but demand substantial resource consumption, often presenting challenges in practical deployment. Among static Webshell detection methods, traditional machine learning algorithms struggle to effectively detect unknown Webshells, while deep learning algorithms rely heavily on computational resources and data volume. Consequently, this study addresses the strengths and limitations of the aforementioned approaches and proposes a Webshell detection method named AST-DF, which combines abstract syntax tree (AST) and the deep forest model. AST-DF initially extracts the abstract syntax tree from PHP code, followed by vectorization using N-Gram and TF-IDF. The classification process is then executed using a deep forest model that integrates concepts from both random forest and deep learning paradigms.

3. Method Architecture

The architecture of AST-DF is shown in Figure 2, which is generally divided into three main parts, namely data preprocessing, feature extraction, and classification. In the preprocessing stage, it mainly includes sample de-duplication and extraction of abstract syntax tree sequences; in the feature extraction stage, the AST sequences obtained in the

previous stage are first subjected to the word-splitting operation, and then the well-split AST sequences are vectorized using the TF-IDF algorithm; and, in the classification stage, the TF-IDF feature vectors are classified by the deep forest model, and the classification result is obtained to achieve the purpose of detecting the Webshells. Each part is described in detail below.

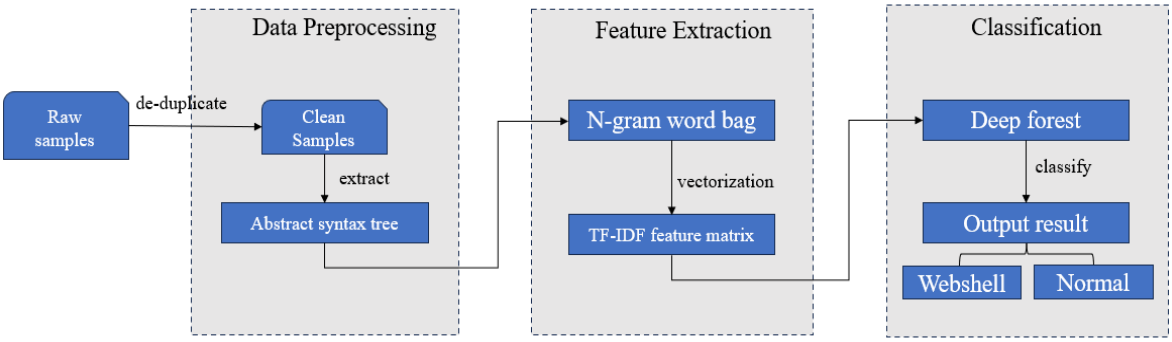

Figure 2. Architecture of AST-DF.

3.1. Data Preprocessing

3.1.1. Sample De-Duplication

When trying to guarantee the uniqueness of a dataset compiled from internet-acquired samples, the issue of a considerable number of duplicate entries often arises. Addressing this challenge necessitates the utilization of the MD5 algorithm [29] for sample de-duplication. The specific approach for eliminating duplicate samples is outlined in Algorithm 1.

Algorithm 1: Sample de-duplication algorithm.

Input Raw samples: code-files
Output: Clean samples: code-files1

```
Initialize the Hash DataSet
for i = 1, 2, 3, . . ., N:
    Hash[i]←MD5 (code-files[i]);
    if Hash[i] belong to Hash DataSet:
        Deleting code-files[i];
    else:
        Adding Hash[i] to Hash DataSet;
        Adding code-files[i] to code-files1;
return Code-files1;
End.
```

Algorithm 1 employs the MD5 algorithm to compute the hash value for each file, subsequently constructing a hash library. The algorithm determines the presence of duplicate samples by verifying whether the file's hash value is contained within the hash library. Upon confirming duplication, the algorithm initiates the deletion of the redundant sample. This procedural algorithm ensures the dataset's sample uniqueness.

3.1.2. Extracting Abstract Syntax Trees

After the processing in Section 3.1.1, the files within the dataset were transformed into a state with no duplicates. In order to further extract the structural and semantic features of the code, the code needs to be extracted from an abstract syntax tree [30]. The AST is a tree-like representation of the syntactic structure of the source code, which is used to analyze and manipulate the code in a more convenient way. The AST also parses the syntactic structure of the code, which provides fine-grained identification of the Webshell feature and enhances accuracy in the detection process.

In the context of PHP, extracting the abstract syntax tree necessitates a comprehensive understanding of the Zend Virtual Machine's (ZendVM) [31] parsing procedures for PHP files, as illustrated in Figure 3. ZendVM engages in both lexical and syntactic analyses during the parsing process. Specifically, in the lexical analysis stage, it meticulously scrutinizes the target PHP file, generating tokens. Subsequently, during syntactic analysis, an AST is crafted. This AST undergoes compilation into Opcode, culminating in the execution of Opcode and the subsequent output of results.

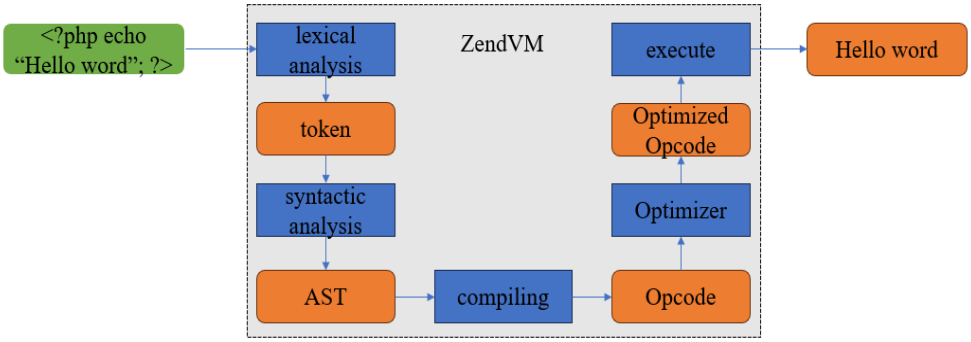

Figure 3. ZendVM parsing PHP file flow.

To extract the AST effectively, one must undertake standard syntactic, lexical, and semantic analyses on PHP files. In this study, the extraction of the abstract syntax tree (AST) for the PHP language is accomplished using PHP-Parser [32]. Post-parsing with PHP-Parser yields a comprehensive syntax tree, wherein not all nodes contribute to the efficacy of Webshell detection. Consequently, it becomes imperative to formulate rules for the extraction of pertinent nodes, with a particular focus on object nodes, to construct the sequence of the abstract syntax tree. To illustrate this, consider a succinct example of a PHP language Trojan horse: '<?php eval($_POST['x']);?>'. The resulting sequence, obtained through PHP-Parser parsing and subsequent rule-based extraction, is depicted in Figure 4.

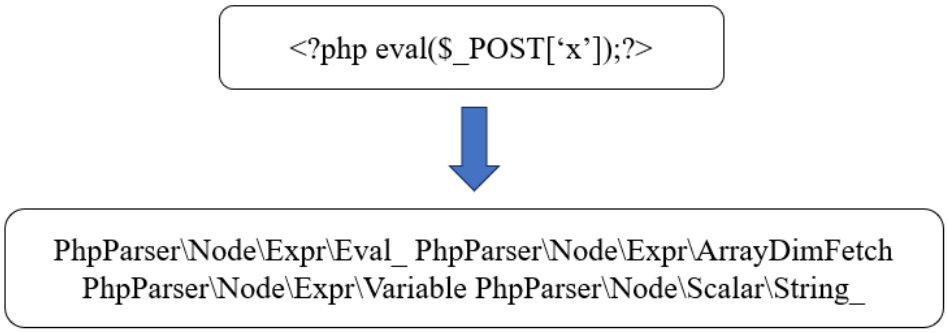

Figure 4. Examples of extracting AST sequences.

3.2. Feature Extraction

The next stage is to sub-word process and vectorize the AST sequences obtained from the previous subsection. After obtaining the AST sequences, the AST sequences are first disambiguated with N-Gram and then vectorized using TF-IDF, where N-Gram [33] is a text analysis and pattern recognition technique used to capture information about a continuous sequence of N elements, usually characters or words. When N is 2, it is a 2-gram, also known as a bigram. In 2-gram, the text is segmented into sequences of two consecutive elements, and an example of segmentation with 2-gram is shown in Figure 5.

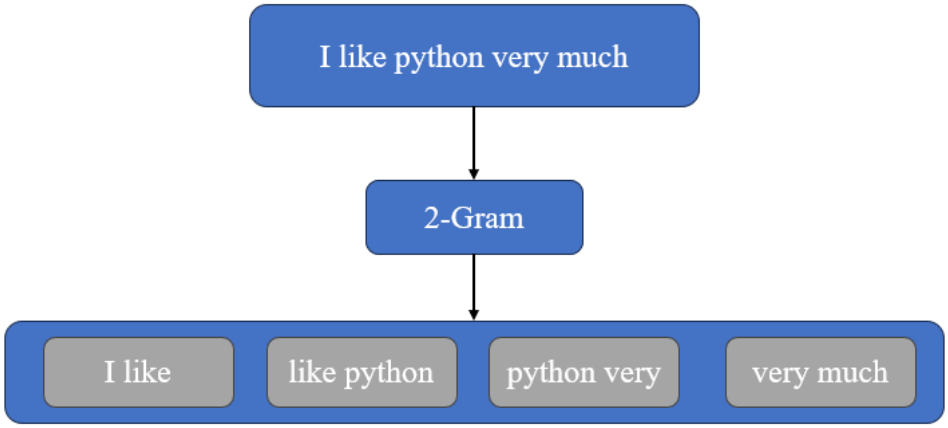

Figure 5. Example of 2-gram.

TF-IDF [34] is a feature representation commonly used in text mining, which combines the frequency of a word in a document (TF) and the inverse document frequency (IDF) in the whole set of documents, where TF is computed as shown in Equation (1), IDF is computed as shown in Equation (2), and TF-IDF is the product of TF and IDF as shown in Equation (3).

$$TF(t, D) = \frac{count(t, d)}{\sum_k count(k, d)} \tag{1}$$

where $count(t, d)$ denotes the number of occurrences of word t in document d, and $\sum_k count(k, d)$ denotes the total number of all words in document d.

$$IDF(t, D) = \log \frac{N}{df(t, D) + 1} \tag{2}$$

where N denotes the total number of documents in the document collection, and $df(t, D)$ denotes the number of documents that contain the word t.

$$TF - IDF(t, D) = TF(t, D) \times IDF(t, D) \tag{3}$$

The TF-IDF model holds widespread utility in domains, such as search engines, text classification, and information retrieval, enhancing the efficacy of text analysis. Within the realm of Webshell detection, a distinctive text classification task, the application of TF-IDF for vectorization, has proven instrumental in elevating detection accuracy. In the context of this study, the abstract syntax tree sequences obtained in the preceding subsection are subjected to vectorization and feature extraction through a combined approach involving N-gram and TF-IDF.

3.3. Classification

Following the crucial steps of feature extraction and vectorization of the abstract syntax tree, the next imperative involves classification using a dedicated model to obtain accurate results. This study opts for the deep forest model [35], a distinctive decision tree ensemble approach renowned for its predictive accuracy, which rivals that of deep neural networks across a diverse array of tasks. A notable advantage lies in the ease of training the deep forest model, attributed to its minimal hyperparameter count. Consequently, this study employs the deep forest model for Webshell detection. The deep forest model comprises two key components: the cascading forest structure and the multi-granularity sliding window. These components play pivotal roles in the model's architecture and will be expounded upon separately in the subsequent sections.

3.3.1. Cascade Forest Structure

The cascade forest structure represents an organizational methodology for multiple classifiers, typically comprising decision trees or random forests, aimed at enhancing overall performance. This strategic arrangement is depicted in Figure 6, providing a visual representation of the conceptual framework. The cascade forest structure serves as a pivotal component in optimizing the collective effectiveness of classifiers within the deep forest model.

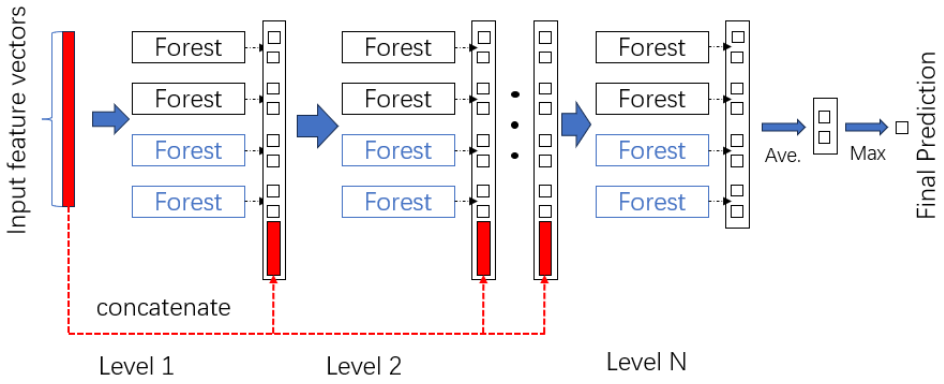

Figure 6. Cascade deep forest structure.

As depicted in Figure 6, the cascade forest structure exhibits a hierarchical organization characterized by sequential relationships, where the output of each layer serves as the input for the subsequent layer. The illustration portrays a binary classification task, aligning with the nature of Webshell detection as a binary classification endeavor. Within each layer, there exist two distinct types of random forest structures: complete-random tree forests (depicted in blue) and random forests (depicted in black). Specifically, two instances of each type are included in the layer configuration. The random forests, encapsulated within black boxes in the figure, consist of 500 trees each, contributing to the hierarchical classification process. Following the traversal of these forests, the output dimensions of the samples become 2. In contrast, the complete-random forests, showcased in the blue boxes, are constructed by random selection of split features and thresholds until the leaf nodes contain samples with identical labels. Moreover, at the output of each layer, the integration of original features and output features from the four forests within that layer ($2 \times 4 = 8$ dimensions) is requisite. This merged output then serves as the input for the subsequent layer, contributing to the cascading architecture of the deep forest model. This strategic ensemble of random and complete-random forest structures within the cascade forest enhances the model's discriminative power.

3.3.2. Multi-Grained Sliding

Multi-Grained sliding refers to the use of a certain size of window and step size for the original feature vector, a full amount of the original features of the sample, split into a number of fixed-length samples, the labels remain unchanged, as shown in Figure 7, the specific process is to set the size of the sliding window and the step size, the expansion of the sample, and then the expansion of the sample of the characteristics of the last is merged in order to increase the randomness of the input features, which can be set to a number of different sliding windows.

In Figure 7, assuming that the input features are 400 dimensions, the size of the window is 100, the step size is 1, and the padding is 0, then 1 sample can be expanded to $(400 - 100)/1 + 1 = 301$ samples. After obtaining the new expanded samples, after going through a random forest and a completely random forest, respectively, 1 sample (expanded to 301 samples) will feature an output vector of $301 \times 2 + 301 \times 2 = 1204$ dimensions, and this 1204-dimension vector will replace the original feature vector as an input to the cascade forest structure.

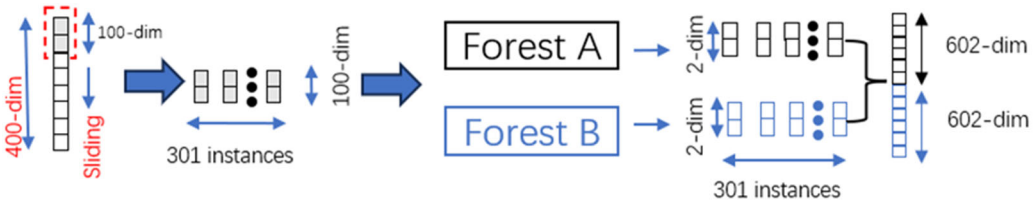

Figure 7. Example of sliding window.

3.3.3. Deep Forest Model

The final deep forest model structure combines the multi-granularity sliding window and cascade forest structure. The overall idea is to first use the multi-granularity sliding window to transform the original input features into higher-dimensional feature representations, which are fed into the different cascade forest structures separately, so that the model essentially forms an integrated re-integrated structure, where each layer of the forest is required to be trained individually, and the input to the model consists of the original features and the outputs of the last forest output of the previous layer, which makes the input of each layer change. The specific structure is shown in Figure 8 below, assuming that the dimension of the feature vectors obtained from Section 3.2 is 400 dimensions, and after three sliding windows with window sizes of 100, 200, and 300, the dimensions of the vectors obtained are 1204-dim, 804-dim, and 404-dim, respectively, and then these vectors are input into the cascade forest structure separately for training, and the vectors of each dimension are input into a different cascade forest structure. There are different cascade forest structures, so three sliding windows will produce three corresponding and independent cascade forest structures; the cascade forest structures are not affected by each other. In the second layer of the input are 1212 dim, 812 dim, and 412 dim, which are the dimensions of the output of the first layer in the cascade forest $2 \times 4 = 8$-dim, respectively, plus the dimensions of the multi-granularity sliding window after being obtained from 1204 dim, 804 dim, and 404 dim. At the end, the maximum of the results of these three cascade forest structures is taken as the final result of the model. This holistic approach, combining multi-granularity feature transformation and cascading forest structures, underscores the model's adaptability and discriminative prowess, particularly in the context of Webshell detection.

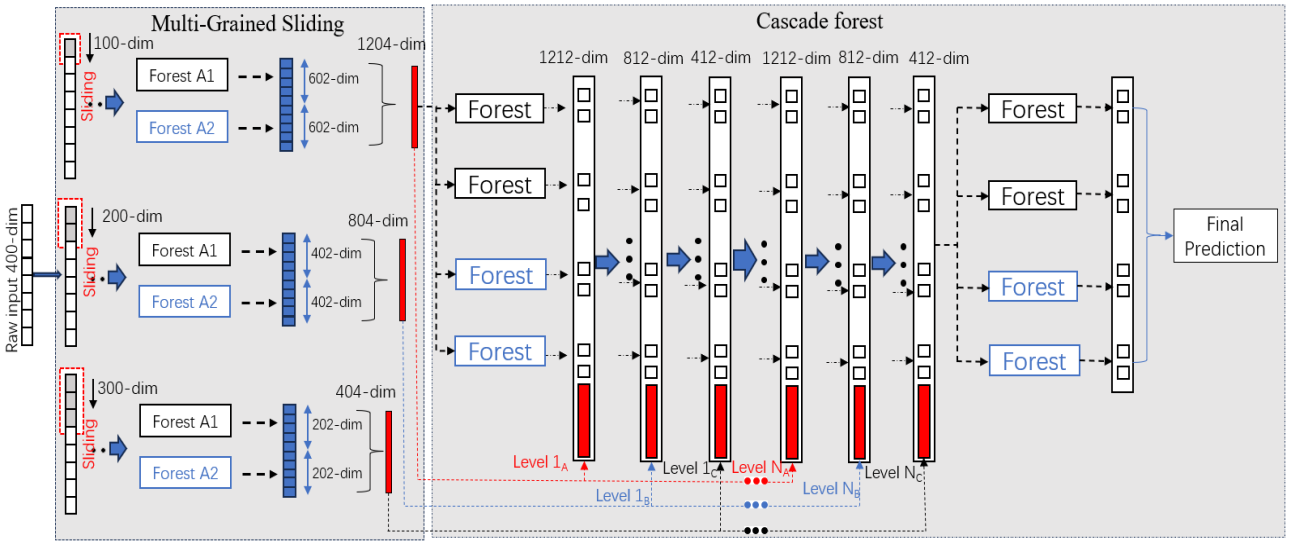

Figure 8. Deep forest structure.

4. Experimental Results and Analysis

4.1. Experimental Environment

The experimental setup leveraged a specified hardware environment, detailed in Table 1. The implementation of this experiment was carried out using the Python programming language. The software employed, along with their respective versions, is systematically outlined in Table 2.

Table 1. Experimental hardware configuration.

Hardware Name	Parameters
Processor	AMD R7 5800U 1.90 GHz
Memory	16 GB
Hard Disk	512GB SSD

Table 2. Software configuration.

Software	Version	Descriptions
Python	3.7.3	experimental language
Sklearn	1.0.2	Machine learning algorithms and evaluation metrics implementation
Deep-forest	0.1.7	Deep forest model implementations
Numpy	1.21.6	matrix operation
Matplotlib	3.5.2	Visualization of the results
PHP	7.3.4	Extracting AST from PHP code

4.2. Dataset

The experimental dataset comprises two categories of samples: Webshell samples and normal samples. The Webshell samples primarily originate from GitHub, as indicated in Table 3, while the normal samples are predominantly sourced from various open-source projects, as detailed in Table 4.

Table 3. Webshell sample sources.

Number	Sources
0	https://github.com/tennc/webshell (accessed on 15 January 2024)
1	https://github.com/xl7dev/WebShell (accessed on 15 January 2024)
2	https://github.com/ysrc/webshell-sample (accessed on 15 January 2024)
3	https://github.com/JohnTroony/php-webshells (accessed on 15 January 2024)
4	https://github.com/tanjiti/webshellSample (accessed on 15 January 2024)
5	https://github.com/DeEpinGh0st/Webshell-bypass-collection (accessed on 15 January 2024)
6	https://github.com/webshellpub/awsome-webshell (accessed on 15 January 2024)
7	https://github.com/tutorial0/WebShell (accessed on 15 January 2024)
8	https://github.com/BlackArch/webshells (accessed on 15 January 2024)
9	https://github.com/x-o-r-r-o/PHP-Webshells-Collection (accessed on 15 January 2024)
10	https://github.com/backdoorhub/shell-backdoor-list (accessed on 15 January 2024)

Table 4. Normal sample sources.

Number	Sources
0	https://github.com/WordPress/WordPress (accessed on 15 January 2024)
1	https://github.com/drupal/drupal (accessed on 15 January 2024)
2	https://github.com/laravel/laravel (accessed on 15 January 2024)
3	https://github.com/joomla/joomla-cms (accessed on 15 January 2024)

Post de-weighting of the collected samples and subsequent extraction of abstract syntax trees, the dataset encompasses a total of 1907 PHP samples representing Webshells

and 11,004 samples representing normal instances. In the experimental phase, the AST-DF model is deployed for the specific task of PHP-type Webshell detection. The dataset is partitioned into training and testing sets at a ratio of 4:1, resulting in 10,329 samples allocated for training and 2852 samples reserved for testing.

4.3. Evaluation Metrics

When using machine learning algorithms for binary classification, four results are produced, and they are determined as positive samples (True Positive, TP), positive samples are determined as negative samples (False Negative, FN), negative samples are determined as negative samples (True Negative, TN), and negative samples are determined as positive samples (False Positive, FP); confusion matrices formed of these four results are shown in Table 5.

Table 5. Confusion matrix.

Actual Result	**Predicted Results**	
	Positive	**Negative**
Positive	TP	FN
Negative	FP	TN

In this study, Webshell files are designated as positive samples, and normal files are positioned as negative samples, and then the model is evaluated using four metrics in machine learning, namely, accuracy, precision, recall, and F1 score, where accuracy denotes the proportion of the number of correctly predicted samples to all the samples; precision denotes the proportion of the samples predicted to be Webshell that are actually Webshell; recall indicates the proportion of samples in Webshell that are correctly predicted as Webshell; and the F1 score is the reconciled average of precision and recall. Since the metrics of precision and recall are contradictory, the F1 score combines the metrics of both, so the F1 score is a good evaluation metric to better represent the performance of the model, and their calculations are shown in Equations (4)–(7).

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{4}$$

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{5}$$

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{6}$$

$$\text{F1 score} = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \tag{7}$$

4.4. Experimental Parameter Setting

In the feature extraction stage, the parameter 'ngram_range' is set to (2,2), indicating the utilization of bigrams. Meanwhile, the max_features parameter is systematically varied, taking values of 2000, 4000, 6000, 8000, and 10,000, respectively. The ensuing variation in accuracy rates corresponding to these different max_features values is graphically depicted in Figure 9. Notably, the highest accuracy rate, reaching 99.61%, is observed when the max_features parameter is set to 6000. Consequently, this optimal value of 6000 is adopted for subsequent stages.

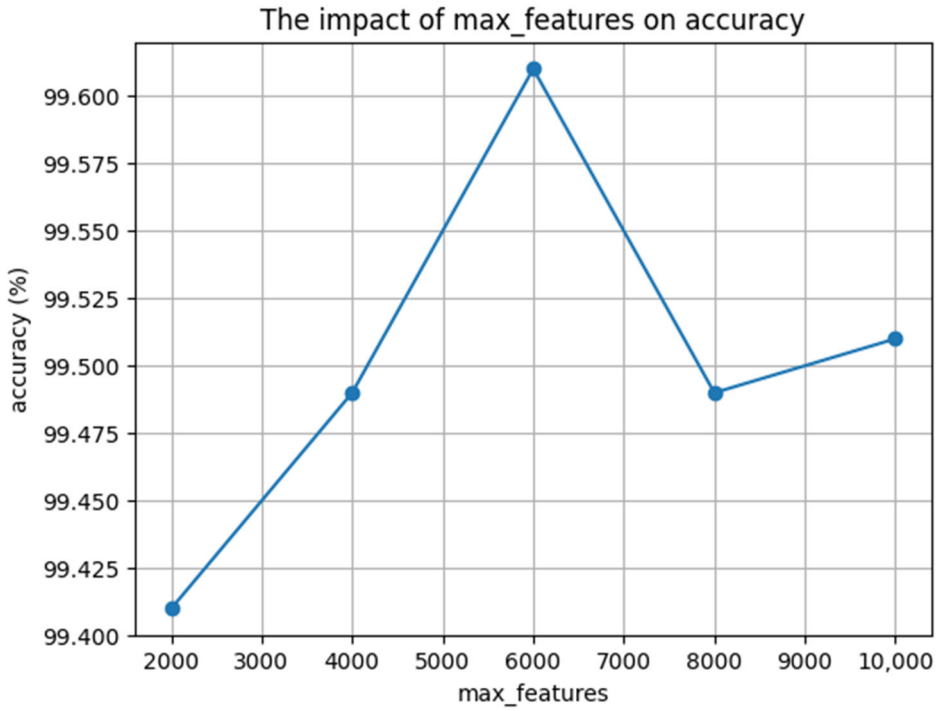

Figure 9. The impact of max_features on accuracy.

During the classification stage, the principal parameters governing the deep forest model are meticulously configured, as delineated in Table 6.

Table 6. Deep forest model parameter settings.

Parameters	Value	Descriptions
max_layers	20	The maximum number of cascade layers in the deep forest.
n_estimators	2	The number of estimators in each cascade layer.
n_trees	100	The number of trees in each estimator.
delta	1×10^{-5}	Specify the threshold on early stopping.

4.5. Analysis of Experimental Results

To substantiate the efficacy of the proposed method, our study conducts experiments tailored to PHP-type Webshell detection. The evaluation employs five distinct machine learning models: Support Vector Machine (SVM), random forest, XGBoost, multilayer perceptron (MLP), and the deep forest model. Each method is individually applied for detection purposes, and the experimental results are presented in Table 7 below. The values bolded in black in the table are the highest values in the indicator.

Table 7. Comparison of detection results of different algorithms.

Methods	Acc (%)	Pre (%)	Recall (%)	F1 Score (%)
SVM	98.03	96.11	90.34	93.14
random forest	99.10	98.70	97.75	98.22
XGBoost	99.54	98.43	98.43	98.43
MLP	98.84	96.82	95.30	96.05
Deep Forest	**99.61**	**99.12**	**99.34**	**99.24**

To elucidate distinctions among various algorithms, the pertinent metrics associated with PHP-type Webshell detection results are graphically represented. Figure 10 below illustrates these metrics, providing a visual depiction of the comparative performance across the different algorithms employed in our study.

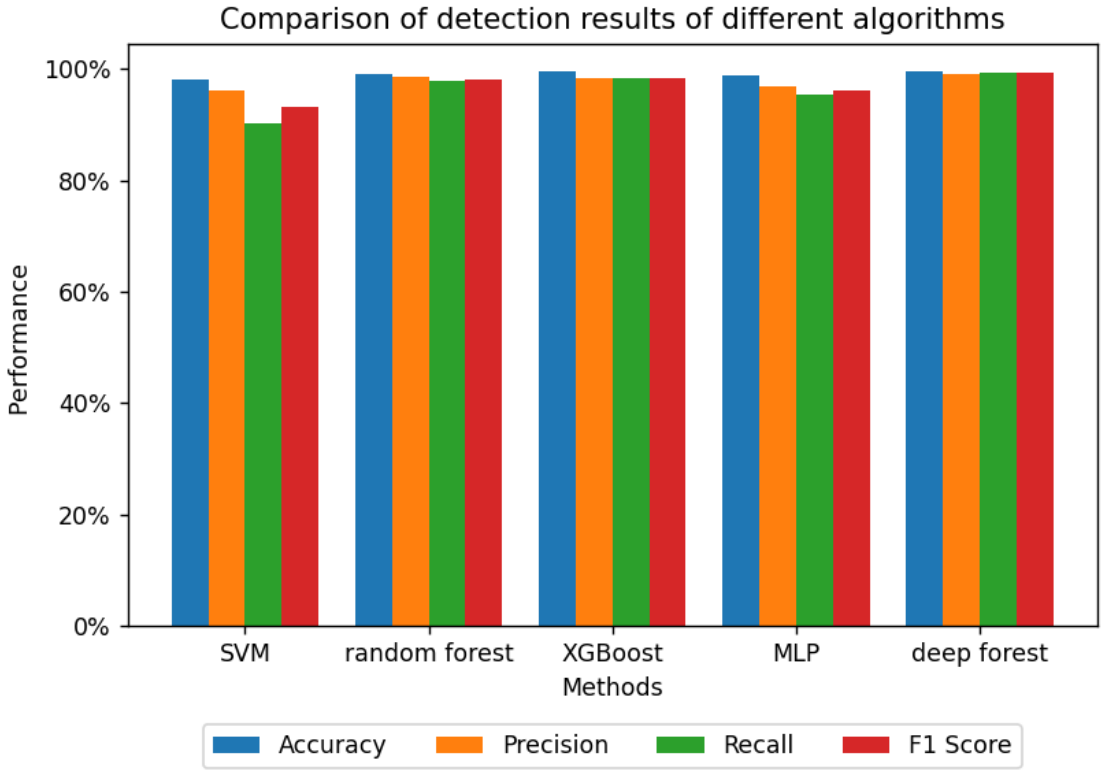

Figure 10. Comparison of detection results of different algorithms.

From Table 7 and Figure 10, it is evident that all algorithms consistently achieve a high accuracy, exceeding 98% when employing both AST and TF-IDF methods for PHP-type Webshell detection. This substantiates the feasibility of utilizing AST for PHP source code extraction in Webshell detection. Notably, the deep forest model outperforms other detection algorithms across various metrics, including accuracy, precision, recall, and F1 score. It is 0.07% higher than the XGBoost algorithm in accuracy; 0.42% higher than the random forest algorithm in precision; and 0.91% and 0.81% higher than the XGBoost algorithm in recall and F1 score, respectively. These findings underscore the effectiveness of the AST-DF approach in detecting PHP-type Webshells.

The detection of PHP-type Webshells is a prominent research focus within the broader field of Webshell detection. Numerous researchers have undertaken extensive investigations in this domain, and we conduct a comparative analysis of methodologies proposed in references [23,28,36,37]. Table 8 presents the performance metrics for each approach. The values bolded in black in the table are the highest values in the indicator.

Table 8. Comparison with Webshell detection methods.

Methods Source	ACC (%)	Pre (%)	Recall (%)	F1 Score (%)
Reference [23]	99.23	97.92	97.65	97.78
Reference [28]	99.36	98.21	98.60	98.40
Reference [36]	97.40	97.20	96.80	97.0
Reference [37]	98.91	**99.25**	95.73	97.46
AST-DF	**99.61**	99.12	**99.34**	**99.24**

In reference [23], the authors leveraged PHP opcode sequences and static features of PHP code as distinctive features. Feature extraction was performed utilizing the fastText model, followed by classification through the random forest algorithm, resulting in an accuracy of 99.23%. The remaining three metrics exceeded 97.60%. In reference [28], the utilization of Word2vec for vectorizing PHP source code, coupled with a bidirectional

GRU network and attention mechanism, yielded an impressive accuracy of 99.3%. Other metrics also surpassed 98.20%. Reference [36] initially extracted the opcode of PHP source code, proceeded to vectorize it using N-gram and TF-IDF, and subsequently employed the plain Bayesian algorithm for classification. This approach achieved a commendable accuracy of 97.4%, with the remaining three evaluation metrics hovering around 97.0%. Reference [37] introduced an integrated learning algorithm that amalgamated a linear regression algorithm, a multilayer perceptron, and a random forest algorithm, combining multiple weak classifiers into a robust single classifier. This approach achieved an accuracy of 99.25% on the experimental dataset.

According to the findings presented in Table 8, AST-DF outperforms the other referenced methods across key metrics, such as accuracy, recall, and F1 score. Despite reference [35] achieving a marginally higher accuracy rate of 99.25%, surpassing AST-DF by 0.13%, it is noteworthy that AST-DF excels in the remaining three metrics. This differential performance underscores the effectiveness of AST-DF in the precise detection of PHP-type Webshells.

In order to be able to comprehensively evaluate the performance of AST-DF, this study compares AST-DF with the current popular Webshell detection tools, such as D-shield, Webshellpub, Cloudwalker, etc., in the task of PHP-type Webshell detection, which support detecting PHP-type Webshells. The detection results of each tool are shown in Table 9. The values bolded in black in the table are the highest values in the indicator.

Table 9. Comparison with Webshell detector tools.

Detector	Version	ACC (%)	Pre (%)	Recall (%)	F1 Socre (%)
D-Shield	V2.1.8.1	98.35	98.90	90.25	94.38
Webshellpub	V1.8.2	93.64	92.39	63.75	75.20
CloudWalker	V1.0.0	93.75	98.37	60.25	74.73
AST-DF	/	**99.61**	**99.12**	**99.34**	**99.24**

According to Table 9, AST-DF achieves the best results in accuracy, precision, recall, and F1 score in the PHP-type Webshell detection task, which is 1.26% higher than D-shield's accuracy; 0.22% higher than D-shield's accuracy; and 9.09% higher than D-shield's accuracy. The F1 score is higher than D-shield by 4.86%. This shows that AST-DF is fully capable of detecting PHP-type Webshells. Although AST-DF performs well in detecting PHP-type Webshells, its scope is limited compared to other detection tools as it can only detect PHP-type Webshells. Future research should focus on expanding the applicability of Webshell detection to cover multiple language types of Webshells.

5. Conclusions

This study takes a detailed look at current Webshell detection techniques and finds that existing methods have limited ability to detect unknown Webshells. In order to enhance the detection capability for PHP-type Webshells and protect network security, we propose a PHP-type Webshell detection method based on abstract syntax tree and the deep forest model, called AST-DF. This approach begins with extracting the abstract syntax tree from PHP code, then applying N-Gram and TF-IDF for feature extraction, and, finally, employing a deep forest model for classification. Experimental findings validate AST-DF's effectiveness in detecting PHP Webshells, achieving accuracy, precision, recall, and F1 scores above 99%. These results confirm AST-DF's capability in overcoming PHP Webshell detection challenges. However, AST-DF has limitations; it is specifically designed for PHP Webshells, making it challenging to adapt to other languages, and it requires periodic updates to its model and feature extraction techniques due to evolving Webshell attack methods, potentially increasing maintenance costs. For future research, we can build more comprehensive and diverse Webshell datasets, develop detection methods capable of identifying multiple language types from source code files, and design more accurate and efficient Webshell detection algorithms by integrating techniques, like deep learning,

natural language processing, and program analysis, to enhance the detection accuracy and generalization capability.

In practical applications, the detection approach outlined in this study can be seamlessly integrated into existing network security systems. Upon initialization, the system generates an initial file hash table. Subsequently, by monitoring file modifications, it identifies changes in the hash values of specific file types or the creation of new files of these types and forwards them for analysis. This enables the immediate detection of malicious activities, such as the uploading of new files or the injection of harmful code by attackers. Furthermore, leveraging this detection technique, a dedicated Webshell detection system can be developed and implemented on servers. This allows users to interact with the system by uploading files for examination. The system analyzes the uploaded files, provides the results, and, thus, extends Webshell detection capabilities externally.

Author Contributions: Conceptualization, C.D.; data curation, C.D.; funding acquisition, D.L.; methodology, C.D.; project administration, D.L.; resources, D.L.; software, C.D. and D.L.; supervision, D.L.; validation, C.D.; writing—original draft, C.D.; writing—review and editing, C.D. and D.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Natural Science Foundation of China (no. 61662004).

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Al-Fedaghi, S. Developing web applications. *Int. J. Softw. Eng. Appl.* **2011**, *5*, 57–68.
2. Kim, J.; Yoo, D.-H.; Jang, H.; Jeong, K. WebSHArk 1.0: A benchmark collection for malicious web shell detection. *J. Inf. Process. Syst.* **2015**, *11*, 229–238.
3. Qian, L.; Zhu, Z.; Hu, J.; Liu, S. Research of SQL injection attack and prevention technology. In Proceedings of the 2015 International Conference on Estimation, Detection and Information Fusion (ICEDIF), Harbin, China, 10–11 January 2015; pp. 303–306.
4. Dahse, J.; Holz, T. Static Detection of {Second-Order} Vulnerabilities in Web Applications. In Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14), San Diego, CA, USA, 20–22 August 2014; pp. 989–1003.
5. Zheng, Y.; Zhang, X. Path sensitive static analysis of web applications for remote code execution vulnerability detection. In Proceedings of the 2013 35th International Conference on Software Engineering (ICSE), San Francisco, CA, USA, 18–26 May 2013; pp. 652–661.
6. Begum, A.; Hassan, M.M.; Bhuiyan, T.; Sharif, M.H. RFI and SQLi based local file inclusion vulnerabilities in web applications of Bangladesh. In Proceedings of the 2016 International Workshop on Computational Intelligence (IWCI), Dhaka, Bangladesh, 12–13 December 2016; pp. 21–25.
7. Le, H.V.; Du, H.P.; Nguyen, H.N.; Nguyen, C.N.; Hoang, L.V. A proactive method of the webshell detection and prevention based on deep traffic analysis. *Int. J. Web Grid Serv.* **2022**, *18*, 361–383. [CrossRef]
8. Yang, W.; Sun, B.; Cui, B. A webshell detection technology based on HTTP traffic analysis. In Proceedings of the Innovative Mobile and Internet Services in Ubiquitous Computing: 12th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS-2018), Matsue, Japan, 4–6 July 2019; pp. 336–342.
9. Tian, Y.; Wang, J.; Zhou, Z.; Zhou, S. CNN-webshell: Malicious web shell detection with convolutional neural network. In Proceedings of the 2017 VI International Conference on Network, Communication and Computing, Kunming China, 8–10 December 2017; pp. 75–79.
10. Hannousse, A.; Yahiouche, S. Handling webshell attacks: A systematic mapping and survey. *Comput. Secur.* **2021**, *108*, 102366. [CrossRef]
11. Lv, Z.-H.; Yan, H.-B.; Mei, R. Automatic and accurate detection of webshell based on convolutional neural network. In Proceedings of the China Cyber Security Annual Conference, Beijing, China, 20–21 July 2018; pp. 73–85.
12. Li, T.; Ren, C.; Fu, Y.; Xu, J.; Guo, J.; Chen, X. Webshell detection based on the word attention mechanism. *IEEE Access* **2019**, *7*, 185140–185147. [CrossRef]
13. Wu, Y.; Song, M.; Li, Y.; Tian, Y.; Tong, E.; Niu, W.; Jia, B.; Huang, H.; Li, Q.; Liu, J. Improving convolutional neural network-based webshell detection through reinforcement learning. In Proceedings of the Information and Communications Security: 23rd International Conference, ICICS 2021, Chongqing, China, 19–21 November 2021; Proceedings, Part I 23; pp. 368–383.
14. Cheng, B.; Guo, Y.; Ren, Y.; Yang, G.; Xu, G. MSDetector: A Static PHP Webshell Detection System Based on Deep-Learning. In Proceedings of the International Symposium on Theoretical Aspects of Software Engineering, Cluj-Napoca, Romania, 8–10 July 2022; pp. 155–172.

15. W3Techs. Available online: https://w3techs.com/ (accessed on 6 April 2024).
16. Zhao, J.; Lu, Y.; Wang, X.; Zhu, K.; Yu, L. WTA: A static taint analysis framework for PHP webshell. *Appl. Sci.* **2021**, *11*, 7763. [CrossRef]
17. Song, Y.; Shen, Y.; Zhang, G. The new INLINE hook technology combination of hard-code technology and independent code injection. In Proceedings of the 2016 7th IEEE International Conference on Software Engineering and Service Science (ICSESS), Beijing, China, 26–28 August 2016; pp. 521–525.
18. Zhang, H.; Guan, H.; Yan, H.; Li, W.; Yu, Y.; Zhou, H.; Zeng, X. Webshell traffic detection with character-level features based on deep learning. *IEEE Access* **2018**, *6*, 75268–75277. [CrossRef]
19. Le, H.V.; Vo, H.V.; Nguyen, T.N.; Nguyen, H.N.; Du, H.T. Towards a Webshell Detection Approach Using Rule-Based and Deep HTTP Traffic Analysis. In Proceedings of the International Conference on Computational Collective Intelligence, Hammamet, Tunisia, 28–30 September 2022; pp. 571–584.
20. Yang, J. A Webshell Detection Model Based on Bayes. In Proceedings of the 2021 2nd International Conference on Computer Communication and Network Security (CCNS), Xining, China, 30 July–1 August 2021; pp. 71–74.
21. Sun, X.; Lu, X.; Dai, H. A matrix decomposition based webshell detection method. In Proceedings of the 2017 International Conference on Cryptography, Security and Privacy, Wuhan, China, 17–19 March 2017; pp. 66–70.
22. Zhang, H.; Liu, M.; Yue, Z.; Xue, Z.; Shi, Y.; He, X. A php and jsp web shell detection system with text processing based on machine learning. In Proceedings of the 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), Guangzhou, China, 29 December–1 January 2020; pp. 1584–1591.
23. Fang, Y.; Qiu, Y.; Liu, L.; Huang, C. Detecting webshell based on random forest with fasttext. In Proceedings of the 2018 International Conference on Computing and Artificial Intelligence, Las Vegas, NV, USA, 12–14 December 2018; pp. 52–56.
24. Chen, T.; Guestrin, C. Xgboost: A scalable tree boosting system. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, 13–17 August 2016; pp. 785–794.
25. Tianmin, G.; Jiemin, Z.; Jian, M. Research on webshell detection method based on machine learning. In Proceedings of the 2019 3rd International Conference on Electronic Information Technology and Computer Engineering (EITCE), Xiamen, China, 18–20 October 2019; pp. 1391–1394.
26. Cui, H.; Huang, D.; Fang, Y.; Liu, L.; Huang, C. Webshell detection based on random forest–gradient boosting decision tree algorithm. In Proceedings of the 2018 IEEE Third International Conference on Data Science in Cyberspace (DSC), Guangzhou, China, 18–21 June 2018; pp. 153–160.
27. Zhou, Z.; Li, L.; Zhao, X. Webshell detection technology based on deep learning. In Proceedings of the 2021 7th IEEE Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing, (HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS), New York, NY, USA, 15–17 May 2021; pp. 52–56.
28. Liu, Z.; Li, D.; Wei, L. A new method for webshell detection based on bidirectional gru and attention mechanism. *Secur. Commun. Netw.* **2022**, *2022*, 3434920. [CrossRef]
29. Rivest, R. *The MD5 Message-Digest Algorithm, 2070-1721*; RFC: Sacramento, CA, USA, 1992.
30. Neamtiu, I.; Foster, J.S.; Hicks, M. Understanding source code evolution using abstract syntax tree matching. In Proceedings of the 2005 International Workshop on Mining Software Repositories, Saint Louis, MO, USA, 17 May 2005; pp. 1–5.
31. Zend. Zend Engine. Available online: https://www.zend.com/ (accessed on 10 February 2024).
32. PHP-Parse. Available online: https://github.com/nikic/PHP-Parser/ (accessed on 10 February 2024).
33. Cavnar, W.B.; Trenkle, J.M. N-gram-based text categorization. In Proceedings of the SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval, Las Vegas, NV, USA, 11–13 April 1994; p. 14.
34. Aizawa, A. An information-theoretic perspective of tf–idf measures. *Inf. Process. Manag.* **2003**, *39*, 45–65. [CrossRef]
35. Zhou, Z.-H.; Feng, J. Deep forest. *Natl. Sci. Rev.* **2019**, *6*, 74–86. [CrossRef] [PubMed]
36. Guo, Y.; Marco-Gisbert, H.; Keir, P. Mitigating webshell attacks through machine learning techniques. *Future Internet* **2020**, *12*, 12. [CrossRef]
37. Ai, Z.; Luktarhan, N.; Zhou, A.; Lv, D. Webshell attack detection based on a deep super learner. *Symmetry* **2020**, *12*, 1406. [CrossRef]