Enhancing the Feature Profiles of Web Shells by Analyzing the Performance of Multiple Detectors

Weiqing Huang, Chenggang Jia, Min Yu, Kam-Pui Chow, Jiuming Chen, Chao Liu, Jianguo Jiang

Chapter 4

ENHANCING THE FEATURE PROFILES OF WEB SHELLS BY ANALYZING THE PERFORMANCE OF MULTIPLE DETECTORS

Weiqing Huang, Chenggang Jia, Min Yu, Kam-Pui Chow, Jiuming Chen, Chao Liu and Jianguo Jiang

Abstract Web shells are commonly used to transfer malicious scripts in order to control web servers remotely. Malicious web shells are detected by extracting the feature profiles of known web shells and creating a learning model that classifies malicious samples. This chapter proposes a novel feature profile scheme for characterizing malicious web shells based on the opcode sequences and static properties of PHP scripts. A real-world dataset is employed to compare the performance of the feature profile scheme against state-of-art schemes using various machine learning algorithms. The experimental results demonstrate that the new feature profile scheme significantly reduces the false positive rate.

Keywords: Web shells, feature profiles, text vectorization, machine learning

1. Introduction

High profile web attacks have highlighted the importance of preventing web application penetrations that serve as springboards for compromising networks [12]. A web shell is often the first step in setting up a backdoor for web application penetration – it is a web script that is placed in a publicly-accessible web server to enable an attacker to obtain web server root permissions and remote control [7].

Accurately detecting web shells in web servers could significantly reduce web application penetration attacks. However, attackers insert hidden functionality in web shells to hinder detection. As a result, evidence pertaining to web shell attacks is difficult to find among the massive

amounts of normal data. Forensic practitioners have to search for web shells manually, a task that is laborious and time-consuming.

Researchers have proposed several methods for detecting malicious web shells. A common approach is to extract features and construct feature profiles that characterize web shells, following which a classification model is developed using a machine learning algorithm.

Liu et al. [5] have proposed a detection model based on convolutional and recurrent neural networks that does not consider the attacker's intentions or require payload sample labeling. Although these learning models may exhibit good performance for specific types of web shells, the models often yield large false positive rates when applied to real-world datasets. Moreover, researchers often ignore the feature profiles and potential behaviors of web shells, and merely view detection as a black-box operation. This makes it difficult to apply and evaluate the detection models in real-world environments. As a result, forensic practitioners have to manually sift through large volumes of data to detect malicious web shells hidden among numerous false positives.

This chapter compares the detection performance of multiple feature profile schemes and machine learning models to identify the reasons for the differences. A general test platform based on real-world web shells collected from public datasets is used to evaluate the performance of the feature profiles and learning models. The experiments reveal that web shell detection performance is affected by feature profiles, text vectorization methods and machine learning models. A novel feature profile scheme is proposed for characterizing malicious web shells based on the opcode sequences and static properties of PHP scripts. The evaluation results demonstrate that the detection method significantly improves malicious web shell classification as well as the ability to detect custom web shell functionality.

2. Related Work

A web shell is a malicious script that attempts to maintain persistent access in an exploited web application [1]. It is assigned to the post-exploitation stage of the attack chain. A web shell does not have any exploitability on its own, but it can be leveraged by attacks such as SQL injection, remote file inclusion and cross-site scripting. Its functions include facilitating persistent remote access, privilege escalation, antivirus software evasion and zombie botnet control.

A web shell can be written in any language that is supported by a web server. Typical examples include PHP, JSP and ASP. Some web shells are tiny, needing only a single line of code whereas others are full-

featured with thousands of lines of code. Well-known web shell families are c99, r57, b374k and barc0de [8].

Wang et al. [14] have used a multi-layer perceptron neural network to detect web shells. They converted sample source code to byte code using a compiler, following which they used bigrams and the term-frequency-inverse-frequency (TF-IDF) statistic to obtain a frequency matrix that was passed to the multi-layer perceptron. Their multi-layer perceptron approach yielded 90% detection accuracy.

Wrench and Irwin [15] have determined the similarity levels between PHP malware samples using four measures to create representative similarity matrices. The malware samples were decoded, the contents of user-defined function bodies and names of user-defined functions were extracted, and file fuzzy hash values were created for similarity analysis.

Yong et al. [16] have employed a deep neural network that detects server-side web shells with good results. Fang et al. [4] have used random forest machine learning with the fastText library to obtain excellent web shell detection results.

Several tools have been developed for web shell detection. One example is CloudWalker for Linux and macOS systems [2]. Web Shell Detector is a PHP script that identifies PHP, CGI(Perl), ASP and ASPX web shells [3]. The `php-malware-finder` tool is designed to crawl filesystems and analyze files against a set of YARA malware identification rules [13]. WebShell.Pub employs traditional features and cloud-based large data dual-engine killing technology [10]. D Shield is an active defense software tool designed for Microsoft IIS systems [11]; it prevents invasions of websites and web servers by applying internal and external protections.

3. Proposed Web Shell Detection Method

This section describes the proposed web shell detection method. It has two components: (i) web shell feature extraction; and (ii) learning model creation.

3.1 Web Shell Feature Extraction

Figure 1 shows the web shell feature extraction process. The model uses PHP opcode sequences of execution path features, opcode sequences of code features and static features to distinguish web shells.

Two types of features are extracted from PHP samples: (i) opcode sequence features; and (ii) static features. The opcode sequences of PHP scripts are obtained using the PHP VLD extension [9] to hook into the Zend engine and dump the opcodes (execution units) of the scripts. Next, the opcode sequences based on execution paths in the PHP scripts

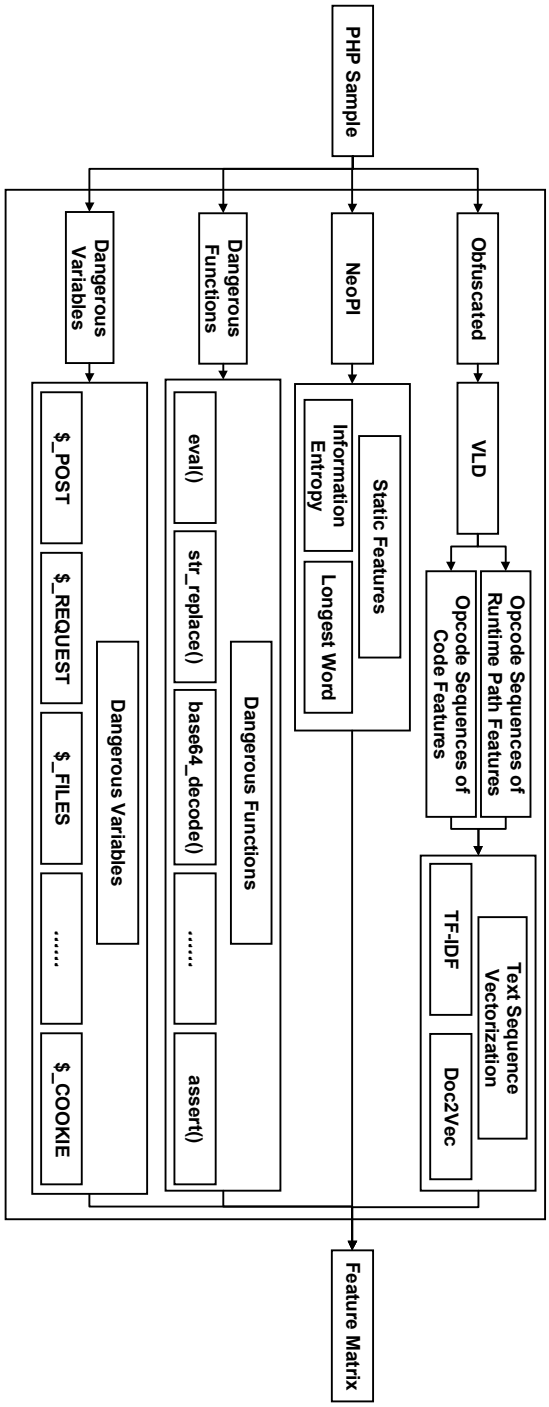

Figure 1. Web shell feature extraction.

```
0.      <?php
1.          echo ‘‘hello world’’;
2.      ?>
```

Figure 2. Simple PHP script.

Table 1. VLD execution results.

Line	Opcode#	Opcode	Branch	Lines	Sop-Eop	Path
1	0	EXT_STMT	0	1-2	0-2	1
	1	ECHO				
2	2	RETURN				

and the opcode sequences corresponding to the code are obtained. TF-IDF and the Doc2Vec tool are used to vectorize opcode sequences of the PHP scripts. Static features such as the longest string, entropy and dangerous functions and variables are also extracted. Finally, the two feature matrices are combined to create a single feature matrix.

Opcode Sequence Features. Opcode arrays generated by a PHP compiler from source code are similar to assembler code generated by a C compiler. However, instead of being directly executed by the CPU, opcode arrays are executed by a PHP interpreter.

The PHP VLD extension is often used for web shell detection [9]. The extension yields two parts. One is the opcode sequences corresponding to the PHP code. The other is the opcode sequences of the execution paths in the code. If only opcode sequences from the first part are considered, interference by certain opcodes causes some PHP code not to be executed, which renders it benign. Therefore, the second part is employed to abstract the opcode sequences.

Figure 2 shows a simple PHP script. Table 1 shows the VLD execution results. The left portion of Table 1 shows the opcodes corresponding to each line of the PHP script. In this portion of the table, Line refers to the line number in the PHP script, Opcode# refers to the opcode number and Opcode refers to the opcode name.

The right portion of the table shows the branches and paths of code execution. Branch refers to the number of the branch, Lines refers to the line numbers in the PHP script corresponding to the branch, Sop-Eop refers to the starting and ending opcode numbers of the branch,

and Path refers to the path number of the branch. In this case, there is only one path, which starts at Line 0 and continues in Lines 1 and 2. The opcode sequence of the code is determined from the Line 0 opcode to Line 2 opcode. The opcode sequence of the path is: EXT_STMT, ECHO and RETURN.

The opcode sequences of the PHP scripts are generated in the form of text. TF-IDF and the Doc2Vec tool are used to vectorize the text to a matrix of fixed dimensions for input to a machine learning algorithm.

Static Features. The static features include: (i) longest string; (ii) information entropy; and (iii) dangerous functions and variables:

- **Longest String:** Web shells maintain stealth using techniques such as encryption, encoding and stitching to obfuscate strings. Normal PHP scripts mostly comprise short strings whereas obfuscated code often contains long strings. The longest string feature is computed as the length of the longest string divided by ten.
- **Information Entropy:** Information entropy is the average rate at which a stochastic source of data produces information. Encryption and compression increase the randomness and information entropy. An obfuscated web shell typically has high information entropy.
- **Dangerous Functions and Variables:** Certain PHP functions are deemed to be high risk. Examples are `eval`, `system`, `assert` and `cmd_shell` that make system calls. Also, `fopen`, `fwrite` and SQL, which can modify files.

3.2 Learning Model Construction

The features extracted from PHP scripts, namely, opcode sequences of code, opcode sequences of runtime paths and static features, are input to TF-IDF and the Doc2Vec tool that vectorize the text for input to a machine learning algorithm. Figure 2 shows the machine learning models employed in the web shell detection framework developed in this research. The models include the support vector machine (SVM), random forest (RF) and k-nearest neighbor (KNN) models.

The dataset was randomly divided to create a training dataset with 70% of the overall data and a testing dataset with the remaining 30% of the data. The training dataset was input to the three learning models to obtain the trained models. The testing dataset was then input to the trained models as well as commercial web shell detection products to evaluate the detection performance.

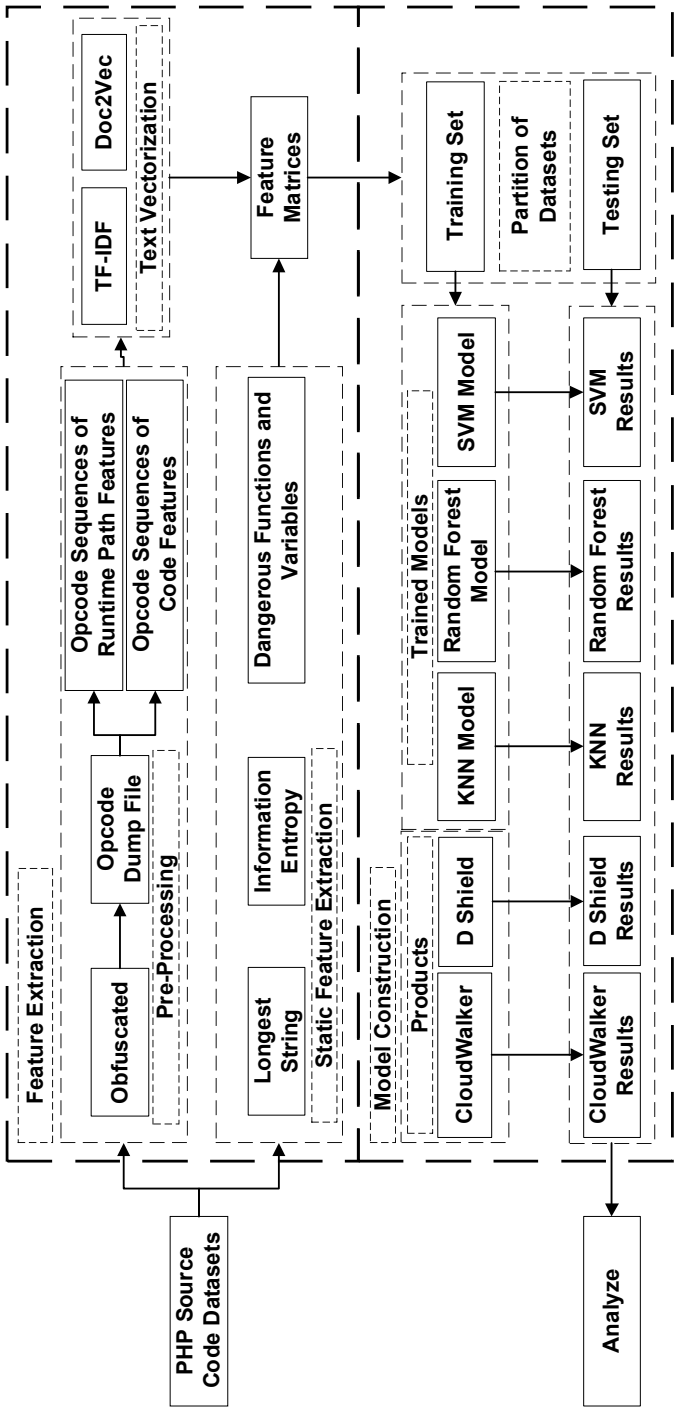

Figure 3. Web shell detection framework.

Table 2. Data sources and samples.

Data Source	Number of Samples	Type
WordPress	4,244	Benign
phpMyAdmin	1,207	Benign
Smarty	213	Benign
Yii	6,202	Benign
PHPCMS	1,207	Benign
GitHub	2,050	Malicious

4. Experiments and Evaluation

This section discusses the data sources and data samples, data processing methods and data used for training and testing. Additionally, it describes three sets of comparative experiments that were conducted to assess the influences of web shell features, text vectorization and learning algorithms (as well as commercial products) on the detection results. The shortcomings of current web shell detection tools with respect to the proposed detection method are also discussed.

4.1 Data Sources and Samples

A total of 13,073 benign PHP scripts and 2,050 malicious PHP scripts were collected. The benign samples came from PHP content management platforms such as WordPress, phpMyAdmin and Smarty. The malicious samples were collected from GitHub projects. Table 2 provides details about the data sources and samples.

4.2 Data Processing

The first step was to compute the hash values of the PHP scripts. Analysis of the hash values revealed that a little over one-half of the malicious samples (1,031 samples) were included in the benign samples. Manual analysis of the 1,031 samples revealed that they were, in fact, benign – this raises questions about the results presented by researchers who have used these datasets [4, 6, 14]. In any case, these 1,031 samples were eliminated to leave only 1,019 malicious samples.

The UnPHP API was employed to deobfuscate the samples. Following this, VLD was used to obtain opcode dumps of the PHP scripts. The opcode sequences were based on the code and execution paths in the dumped files. The opcode sequences were input to TF-IDF and Doc2Vec to obtain two types of feature vectors. Meanwhile, NeoPI was employed to obtain the static features of the samples and count the numbers of

Table 3. Dataset summary.

Category	Property	Value
Dataset	Number of samples	12,309
	Number of benign samples	11,397
	Number of malicious samples	912
Number of Input Features	TF-IDF	181 + 23 = 214
	Doc2Vec	10 + 23 = 33
Training Set (70%)	Number of benign samples	7,979
	Number of malicious samples	646
Testing Set (30%)	Number of benign samples	3,418
	Number of malicious samples	266

malicious functions and variables in each sample. Since some scripts could not be analyzed by NeoPI, the number of malicious samples was reduced to 912. Finally, the vectorized and static features were combined to obtain the final sample features.

Table 3 provides details about the final dataset, numbers of input features, and the training and testing datasets.

4.3 Evaluation of Feature Sets

This section discusses the effects of input features on the classification results. In order to compare the classification results for different feature sets, TF-IDF was used for text vectorization and random forest (RF) was selected as the learning model.

Figure 4 shows the detection results obtained using two types of opcode sequences (path_seq and code_seq) and static features. The path_seq feature yielded the best accuracy, recall and F1 score metrics whereas the static features yielded poor results for these metrics.

Table 4 shows the true positive (TP), false negative (FN), false positive (FP) and true negative (TN) values for combinations of opcode sequence and static features with TF-IDF vectorization compared with static features alone. Note that using an opcode sequence feature with static features produced better results than using only static features.

Analysis of the false negative samples revealed that the opcode sequence features can distinguish some malicious samples with static features that are not obvious. This was especially noticeable in the case of custom malicious functions used to create backdoors; examples include database write operations and file entry operations. A normal database operation directly stores the data from a form to the database. How-

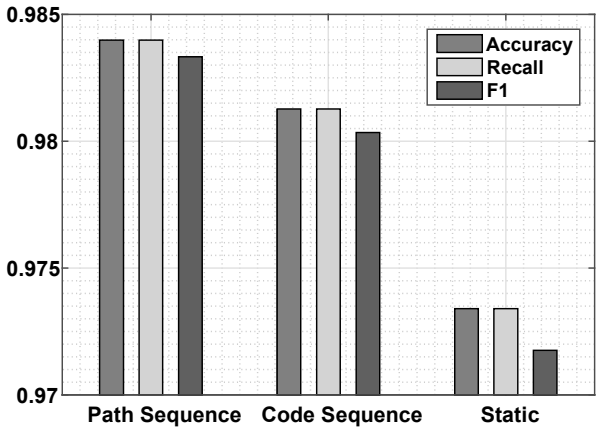

Figure 4. Evaluation of feature sets.

Table 4. Detection performance for various feature sets.

Feature Sets	TP	FN	FP	TN
path_seq + static + TF-IDF + RF	3,410	8	51	215
code_seq + static + TF-IDF + RF	3,409	9	68	198
static + RF	3,400	18	80	186

ever, a malicious database operation decrypts the data from the form and then operates on the database based on the decrypted string. When only static features were used, the malicious operations were classified as normal because there were no malicious features aside from the decryption function. For these reasons, 35 malicious samples could be detected using the static features alone.

The 26 samples that were only detected using opcode sequence features were also analyzed. Most of the samples could not be detected using only static features because they employed custom malicious functions instead of common malicious functions. This shows the relative advantage of using opcode sequence features based on runtime paths.

Table 5 shows the numbers of malicious functions that were detected when opcode sequence and static features were used in combination, but were not detected when static features were used alone. The samples are divided into six categories based on the malicious functions: (i) command line; (ii) file read and write; (iii) file search; (iv) database backdoor; (v) encrypted communication; and (vi) password acquisition.

Table 5. Sample misclassification (opcode + static features vs. static features).

Category	**Malicious Functions in Misclassified Samples**
Command Line	14
File Read and Write	12
File Search	4
Database Backdoor	2
Encrypted Communication	9
Password Acquisition	2

4.4 Evaluation of Text Vectorization Methods

Three comparative experiments were conducted to verify the suitability of the TF-IDF and Doc2Vec text vectorization methods. In order to compare the two text vectorization methods, the opcode sequences of execution path features and static features were combined to create the input features, and random forest was used as the learning model. TF-IDF and two Doc2Vec versions, Doc2Vec10 and Doc2Vec181, that generated ten and 181 vector parameters, respectively, were evaluated.

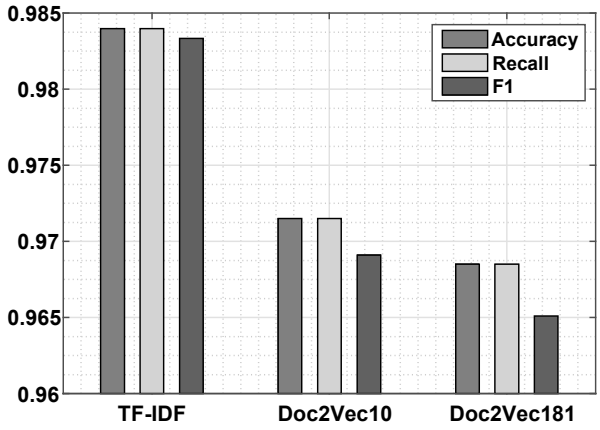

Figure 5. Evaluation of text vectorization methods.

Figure 5 demonstrates that TF-IDF performed better than both versions of Doc2Vec in terms of accuracy, recall and the F1 score.

Table 6. Detection performance for various text vectorization methods.

Vectorization Method	TP	FN	FP	TN
TF-IDF	3,410	8	51	215
Doc2Vec10	3,409	9	93	173
Doc2Vec181	3,410	8	108	158

Table 6 shows that the true positive rates are similar for TF-IDF and the two Doc2Vec versions, but the false positive and true negative rates are significantly lower for both Doc2Vec versions.

The comparative experiment using Doc2Vec10 and Doc2Vec181 to generate ten vectors and 181 vectors, respectively, was conducted to prove that static features have higher weights among all the features. The results in Figure 5 and Table 6 indicate that the presence of too many Doc2Vec vectors weakened the proportion of static features, leading to a decrease in detection performance.

Table 7. Sample misclassification (TF-IDF vs. Doc2Vec).

Category	Malicious Functions in Misclassified Samples
Command Line	17
File Read and Write	14
File Search	4
Database Backdoor	11
Encrypted Communication	9
Password Acquisition	1

Table 7 shows the numbers of malicious functions that were detected when TF-IDF was used, but were not detected when Doc2Vec was used. The Doc2Vec tool was unable to detect considerable numbers of command line (long samples), file read and write, and database backdoor functions. This is because the method underlying Doc2Vec pays more attention to context than the TF-IDF method. Since only one path in the code sequences of a malicious sample may be malicious, there is considerable interference by non-malicious paths on the features. This also explains why researchers have suggested that the Doc2Vec detection performance is below par [6].

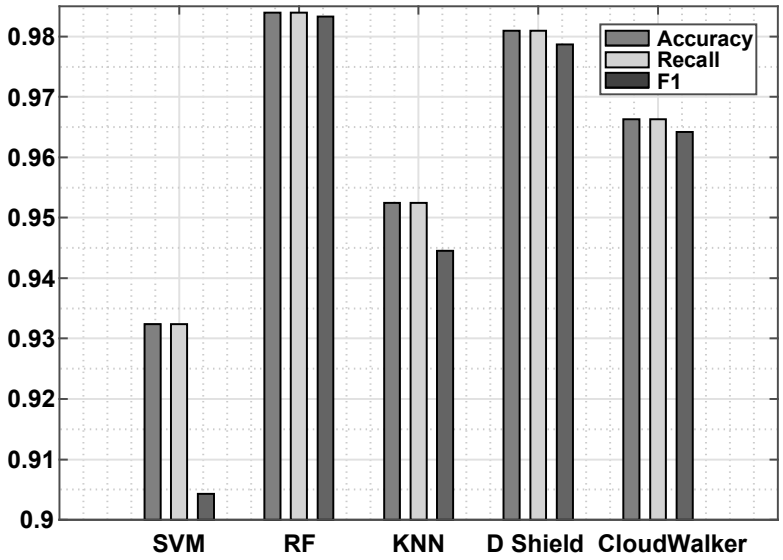

Figure 6. Evaluation of detection algorithms and commercial products.

4.5 Evaluation of Algorithms and Products

The experiments described above demonstrated that opcode sequence features combined with static features and TF-IDF are the best combination for detecting malicious web shells. This section discusses the detection results obtained for various machine learning algorithms with the best feature set combination and TF-IDF along with the detection results obtained using two commercial web shell detection products. The machine learning algorithms included the support vector machine (SVM), random forest (RF) and k-nearest neighbor (KNN) algorithms. The commercial products included D Shield [11] and CloudWalker [2].

The results in Figure 6 demonstrate that the random forest algorithm based web shell detection solution yielded much better detection performance compared with the support vector machine and k-nearest neighbor algorithm based solutions in terms of accuracy, recall and the F1 score. This is because the random forest algorithm, which is an ensemble learning model based on decision trees, has better generalization ability than single models like the support vector machine and k-nearest neighbor algorithms. Figure 6 also shows that the random forest algorithm based solution was slightly better than the D Shield product and moderately better than CloudWalker in terms of accuracy, recall and the F1 score.

Table 8. Detection performance of various algorithms and commercial products.

Algorithms and Products	TP	FN	FP	TN
SVM Algorithm	3,417	1	248	18
RF Algorithm	3,410	8	51	215
KNN Algorithm	3,410	18	157	109
D Shield Product	3,410	8	108	158
CloudWalker Product	3,392	26	98	168

Table 8 shows that the random forest algorithm based solution has the best overall performance compared with the other two learning algorithm based solutions and the two commercial products. Specifically, the random forest algorithm based solution has the lowest combination of false negative and false positive values.

Table 9. Sample misclassification (random forest solution vs. D Shield).

Category	Malicious Functions in Misclassified Samples
Command Line	7
File Read and Write	9
File Search	3
Database Backdoor	1
Encrypted Communication	7
Password Acquisition	1

Table 9 shows the numbers of malicious functions that were detected when the random forest algorithm based solution was used, but were not detected by D Shield. Analysis of these samples revealed that D Shield is poor at detecting web shells with custom malicious functions. The results also reveal that using opcode sequences of execution path features enhance malicious web shell detection.

However, 15 malicious samples that were detected by D Shield were missed by the random forest algorithm based solution. Analysis of these samples revealed that the random forest based solution was hindered by the inclusion of functions such as `eval()` and `phpinfo()`. These

functions also appeared in a small number of positive samples, which may be the reason for their misclassification.

5. Conclusions

This chapter has proposed a malicious web shell detection method that leverages opcode sequence and static features of PHP scripts along with text vectorization and machine learning. Experiments using a general web shell detection framework relying on real-world data collected from public datasets reveal that the detection performance is affected by feature profiles, text vectorization methods and machine learning models. The experimental evaluations demonstrate that using the combination of opcode sequence and static features along with TF-IDF vectorization and the random forest machine learning algorithm outperforms other machine learning algorithm based solutions as well as the D Shell and CloudWalker commercial web shell detection products. In particular, the low false positive rate renders the proposed method useful and efficient in forensic investigations.

Acknowledgement

This research was supported by the Natural Science Foundation of China under Grant no. 61402476 and by the National Key R&D Program of China under Grant no. 2017YFB0801900.

References

[1] Acunetix, An Introduction to Web-Shells, London, United Kingdom (`www.acunetix.com/websitesecurity/introduction-web shells`), 2016.

[2] Chaitin Tech, CloudWalker Platform, GitHub (`github.com/chai tin/cloudwalker`), March 7, 2020.

[3] M. Emposha, PHP-Shell-Detector, GitHub (`github.com/emposha/ PHP-Shell-Detector`), October 5, 2015.

[4] Y. Fang, Y. Qiu, L. Liu and C. Huang, Detecting web shells based on random forest with fastText, *Proceedings of the International Conference on Computing and Artificial Intelligence*, pp. 52–56, 2018.

[5] H. Liu, B. Lang, M. Liu and H. Yan, CNN and RNN based payload classification methods for attack detection, *Knowledge-Based Systems*, vol. 163, pp. 332–341, 2019.

[6] Z. Lv, H. Yan and R. Mei, Automatic and accurate detection of web shells based on convolutional neural networks, *Proceedings of the China Cyber Security Annual Conference*, pp. 73–85, 2018.

[7] MITRE Corporation, Web Shell, Bethesda, Maryland (attack.mitre.org/techniques/T1100), 2019.

[8] T. Moore and R. Clayton, Evil searching: Compromise and recompromise of Internet hosts for phishing, *Proceedings of the International Conference on Financial Cryptography and Data Security*, pp. 256–272, 2009.

[9] D. Rethans, More Source Analysis with VLD (derickrethans.nl/more-source-analysis-with-vld.html), February 19, 2010.

[10] ShellPub.com, Webshell.Pub, Beijing, China (www.shellpub.com), 2020.

[11] Shenzhen Di Element Technology, D Shield, Shenzen, China (www.d99net.net), 2020.

[12] O. Starov, J. Dahse, S. Ahmad, T. Holz and N. Nikiforakis, No honor among thieves: A large-scale analysis of malicious web shells, *Proceedings of the Twenty-Fifth International Conference on World Wide Web*, pp. 1021–1032, 2016.

[13] J. Voisin, php-malware-finder, GitHub (github.com/nbs-system/php-malware-finder), May 26, 2020.

[14] Z. Wang, J. Yang, M. Dai, R. Xu and X. Liang, A method for detecting web shells based on multi-layer perception, *Academic Journal of Computing and Information Science*, vol. 2(1), pp. 81–91, 2019.

[15] P. Wrench and B. Irwin, Towards a PHP web shell taxonomy using de-obfuscation-assisted similarity analysis, *Proceedings of the Information Security for South Africa Conference*, 2015.

[16] B. Yong, X. Liu, Y. Liu, H. Yin, L. Huang and Q. Zhou, Web behavior detection based on deep neural networks, *Proceedings of the IEEE SmartWorld, Ubiquitous Intelligence and Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People and Smart City Innovation Conferences*, pp. 1911–1916, 2018.