SAWD: Structural-Aware Webshell Detection System with Control Flow Graph

Junmin Zhu[†§], Yizhao Yao[‡§], Xianwen Deng[†§], Yaoguang Yong[†], Yanhao Wang[††],
Libo Chen[†], Zhi Xue[†*], and Ruijie Zhao[†*]
[†]Shanghai Jiao Tong University, Shanghai, China
[‡]Tencent Technology (Shanghai) Company Limited, Shanghai, China
[††]QI-ANXIN, Beijing, China
[§]Equal Contribution [*]Corresponding Authors

Abstract

With the increasing prevalence of web servers, protecting them from cyber attacks has become a crucial task for online service providers. Webshells, which are backdoors to websites, are commonly used by hackers to gain unauthorized access to web servers. However, traditional methods for detecting webshells often fail to produce satisfactory results due to the use of obfuscation or encryption to conceal their characteristics. In recent years, webshell detection methods based on deep learning (DL) have received significant attention, but they struggle to preserve the syntax and semantic information contained in the source code. In this paper, we propose a structural-aware webshell detection system to address these problems, denoted as SAWD. Specifically, we first generate the control flow graph (CFG) with syntax and semantic information from the PHP source code. Then, we leverage CFG to build our graph representation, which consists of the adjacency matrix and keywords-based basic block features. Finally, based on our graph representation, we adopt convolutional neural networks (GCN) combined with graph pooling to detect webshells more efficiently. Experimental results demonstrate that our method outperforms state-of-the-art webshell detection systems on the collected dataset.

***Index Terms*—Webshell detection, deep learning, control flow graph, graph neural networks.**

I. Introduction

With the development of the internet, web servers have become a prime target for hackers. Hackers often use webshell, a web script that contains a malicious code fragment, to launch web attacks for system intrusion. Once the web server is compromised, a vast amount of sensitive user information stored in it will be stolen, causing irreparable damage to the service provider. Thus, it is critical to design an effective webshell detection system to protect system security.

Currently, webshell detection systems are divided into three categories, namely traffic-based detection, log-based detection, and file-based detection. Traffic-based detection involves real-time monitoring of server communication traffic and determining if it is a request communicating with a webshell through pre-set rules [1]. Unfortunately, it is inefficient to detect webshell behavior from large-scale traffic. Log-based detection examines whether the HTTP requests and responses recorded on the website contain malicious behavior for webshell detection [2]. However, this system can only audit the behavior during or after an attack, which means the system has been hacked. Compared with the above two solutions, File-based detection is more efficient for finding potential threats before attackers launch the attack. It analyzes the code within program files to extract functional and logical characteristics for detecting webshells.

Traditional file-based webshell detection methods mainly rely on building feature libraries based on file attributes (e.g., file name, file creation time, file modification time, and high-risk functions), but they may not be applicable to webshell files that have been encrypted or obfuscated, which are beyond the scope of the established feature libraries. Researchers then suggest using statistical features like information entropy, longest word, and overlap index to identify obscured webshell files. However, they have limited applicability as well as low detection performance. Another mainstream webshell detection method is to analyze the data dependencies between variables through syntax and semantic analysis. Since it requires manual setting of pollution sources, pollution recipients,

DOI reference number: 10.18293/SEKE2023-205

and pollution propagation rules, it may become difficult to detect the webshell using new PHP features. Recently, webshell detection methods based on deep learning (DL) have received significant attention. They first convert the source code to the opcode sequence, and then adopt deep neural networks for feature extraction. However, this method often disregards the operands within the opcode, thereby overlooking critical information contained therein. So this approach struggles to fully capture the syntax and semantic information of PHP source code.

In this paper, to solve the above challenges, we propose a **S**tructural-**A**ware **W**ebshell **D**etection system (SAWD), which generates graph representation for modeling PHP source code and leverages DL-based graph networks for feature extraction. Specifically, we first generate the control flow graph (CFG) that can preserve the syntax and semantic information of PHP source code. Then, we convert the CFG into the adjacency matrix and extract the feature of the basic block by counting the most frequent keywords, which plays a significant role in detecting obfuscation and encryption behavior. At last, we use graph convolutional neural networks (GCN) combined with MinCutPool as our structural-aware model for webshell detection. GCN is used to extract features from a topological graph, then we utilize the graph pooling method to cluster the nodes based on the similarity of the basic blocks that perform the same function in the CFG, thus improving the generalization ability. The major contributions of the proposed work are three-fold:

- To generate the effective graph representation with the syntax and semantic information, we first convert the PHP source code to CFG. Then, we adopt the keywords-based method to represent each basic block and the adjacency matrix to capture the relationship between basic blocks.
- We develop a structural-aware webshell detection model using GCN combined with graph pooling, which can leverage our graph representation for more efficient code analysis. To our best knowledge, this is the first investigation in this direction.
- We evaluate SAWD on our collected dataset with three webshell types i.e., big trojan, small trojan, and one-word trojan. Experimental results demonstrate that our method outperforms state-of-the-art webshell detection systems.

II. Related Work

To efficiently detect encrypted webshell, researchers proposed to use file-based webshell detection systems. Some work identifies obfuscated and encrypted content in scripts (e.g., NeoPI [3]), but the design of statistic features in their methods relies heavily on expert knowledge. In recent work, more and more studies adopt deep learning algorithms for webshell detection, which can be categorized based on the features used as source code-level methods, opcode-level methods, and AST-level methods. Li et al. [4] utilized Word2Vec for the vectorization of PHP source code and gated recurrent unit (GRU) for effective detection. Zhao et al. [5] converted PHP code into opcode sequences and incorporated TF-IDF-based weighted processing techniques, utilizing an XGBoost model for classification. Kang et al. [6] proposed an RF-GBDT model that employed an improved TFIDF-chi feature to extract webshell opcode features. Additionally, they combined statistical features and opcode sequences of PHP files to enhance detection efficiency. Although using opcodes as features can reduce the impact of techniques such as obfuscation on detection performance, they do not preserve syntax and semantic information present in the original code. Li et al. [7] presented Shellbreaker which uses static analysis and machine learning techniques to extract features from PHP scripts for detection. However, this approach is vulnerable to failure when identifying new webshells, especially when attackers use advanced encoding strategy (e.g., letter slicing). Cheng et al. [8] proposed MSDetector to parse PHP scripts into an AST and extract lexical tokens. Although using AST as a feature effectively preserves syntax and lexical information present in source code, semantic information is challenging to retain completely.

In the field of program analysis, graph-based methods have become increasingly common. Reps [9] first proposes to convert some program analysis problems into graph reachability problems, and solve them using graph reachability algorithms. Yamaguchi et al. [10] introduced the code property graph, which combines the properties of abstract syntax trees, control flow graphs, and program dependence graphs to create a novel representation of source code. Backes et al. [11] applied this approach to PHP and used taint analysis to discover vulnerabilities in PHP code. However, there is currently no work on detecting webshells based on graphs. Additionally, graph neural networks can utilize the information of nodes and edges in a graph to learn representations of nodes, greatly enhancing their ability to preserve the complete syntax and semantic information of source code. All these methods have inspired us to adopt graph-based methods and graph neural network models to address the problem of incomplete preservation of syntax and semantic information in previous work.

III. Method

We describe the detailed designs in this section. The overview of our proposed method is illustrated in Fig. 1.

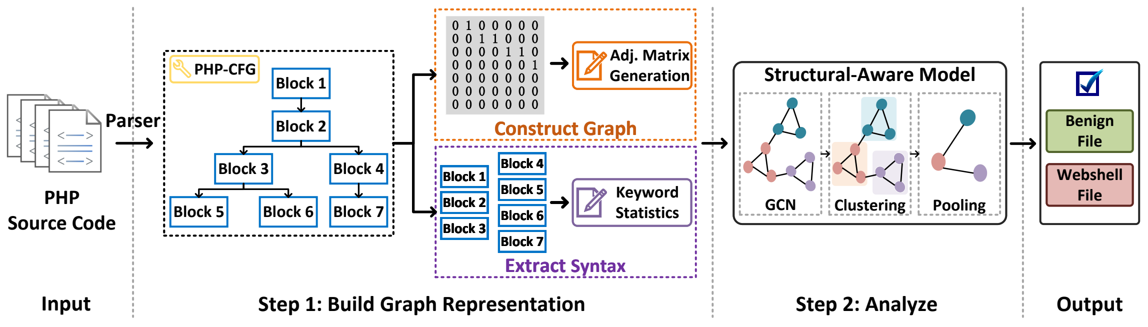


Fig. 1. Overview of the proposed method for webshell detection.

A. Graph Representation of PHP Source Code

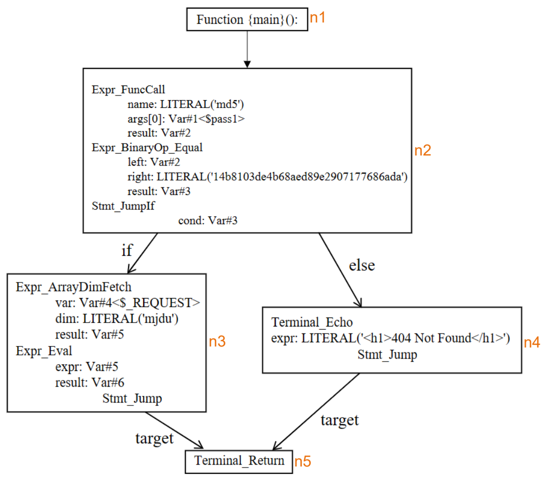
1) Graph Construction: We employ PHP-CFG, an open-source project developed and maintained by Anthony Ferrara, to establish the CFG from the PHP source code. It first parses PHP code into an Abstract Syntax Tree (AST), which contains all the syntax information in a program. To further capture the semantic information, PHP-CFG then iterates through the AST nodes to group adjacent AST nodes in the same code branch into a basic block. Finally, all the blocks are connected by directed edges according to the execute sequence and conditional jumps, and the directed CFG is established, which precisely provides a clear representation of the syntax and semantic information. The variables and function calls in the basic code block represent the syntax information, while the connection relationships between blocks, i.e., the execution sequence and conditional jumps, represent the semantic information.

Fig. 2 illustrates an example of a webshell and its corresponding CFG, where the CFG consists of four nodes and three edges. Each node corresponds to a basic code block, and each edge represents a conditional or loop relationship in the PHP code. For the PHP source code, PHP-CFG first groups the conditional AST node and its preceding nodes in the same code branch into a basic block. In addition to normal basic blocks, a main code block is automatically created as the entry point during the initial phase. Starting from the main code block, PHP-CFG traverses all blocks and establishes ”if-else” relationships as edges for connecting these code blocks.

We then convert the directed CFG into the digitized adjacency matrix, which enables us to quickly identify the entry and exit points of a CFG, as well as its loops and branches. Moreover, the symbolic adjacency matrix can be used to perform various graph-based algorithms. Specifically, we first generate an $N \times N$ zero matrix, where N is the total number of blocks in a CFG. This matrix serves as a template for capturing the relationship between

```
<?php
if(md5($pass1)=='14b8103de4b68aed89e2907177686ada'){
    eval($_REQUEST['mjdu']);
}
else{
    echo "<h1>404 Not Found</h1>";
}
?>
```

(a) Source code of the webshell case.



(b) CFG of the wbeshell case.

Fig. 2. Example of a PHP webshell source code and its corresponding CFG.

basic blocks. If there is a directed edge between any two basic blocks, for example from block m to block n, we set the corresponding entry in the matrix to 1. To illustrate this, let us continue to consider the CFG in Fig. 2, which consists of five block nodes labeled n1, n2, n3, n4, and n5. Note that n2 has three adjacent blocks: n1, n3, and n4. The control flow goes from n1 to n2, then from n2 to either n3 or n4. To represent these connections, we set entries (1, 2), (2, 3), and (2, 4) in the adjacency matrix to 1, while (2, 1) remains 0. Similarly, we apply the same procedure to all other basic blocks in the CFG. The conversion result is shown in Fig. 1 Step 1. By doing so, we obtain an adjacency matrix to visualize the control flow relationships among the basic blocks. We denote the $N \times N$ adjacent matrix as $\mathbf{A}_{N \times N}$.

2) Syntax Feature Extraction: The syntax feature of a basic code block (i.e., a node in CFG) is extracted according to AST node types, variables, and function calls. In CFG, each basic block consists of one or more AST nodes combined together. There are approximately 140 distinct nodes in PHP syntax, which can be categorized into three main groups:

- The statement node, a language structure that does not produce a value and is not capable of occurring within an expression, such as a class definition;
- The expression node, a language structure that generates a value and can therefore be utilized within other expressions;
- The scalar node, which represents scalar values, like 'string' or magic constants, like '__FILE__'
- In addition to these three groups there exists nodes that do not belong to either of these categories, such as names and call arguments.

Some special node types, parameters, and global variables appear frequently in webshells, which is an important symbol to identify these web scripts. Therefore, we extract basic features for each code block by counting the occurrences of some key events, including:

1) The number of some AST node types in the block, such as Expr_FuncCall.
2) The number of parameters in some AST nodes in the block, such as var, dim, and result.
3) The number of global variables in the block, such as $_GET, $_POST, $_REQUEST, which is particularly crucial for webshells.
4) The number of some common literal types of keywords in the block. Literal is a type that represents a literal value, such as a string ("mjdu" in Fig. 2), a number, a Boolean value, or null.

By computing these key events, we can capture important characteristics of a code block and leverage them for various tasks such as code classification, similarity analysis, or anomaly detection. The syntax feature of the k-th basic code block (i.e., the k-th node feature in CFG) is denoted as F_k.

B. Structural-Aware Model

We use a GCN [12] combined with MinCutPool [13] pooling as our structural-aware model for webshell detection. Let the CFG be represented by a tuple $G = \{\mathcal{V}, E\}$, $|\mathcal{V}| = N$, where $\mathcal{V}$ represents the block set and E represents the connections in the CFG. Each node is associated with a vector attribute F_k, i.e., the syntax feature of a basic code block. A graph is characterized by its adjacency matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ and the node features $\mathbf{F} \in \mathbb{R}^{N \times D}$.

The structure-aware model is stacked by GCN layers with MinCutPool layers. Let $\hat{A} = \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{\frac{1}{2}}$ be the symmetrically normalized adjacency matrix, where $\tilde{A} = A + I$ means modifying the graph by adding self-loops and $\tilde{D}$ is the degree matrix of $\tilde{A}$. Assuming that the input of the l-th GCN layer is $H^l \in \mathbb{R}^{N \times D}$, the output is computed as:

$$H^{l+1} = \sigma(\hat{A} H^l W^l), \tag{1}$$

where σ is the activation function and W^l is a learnable parameter. Note that we direct use the syntax feature $\mathbf{F} \in \mathbb{R}^{N \times D}$ as the input of the first GCN layer.

The essence of the GCN layer is to allow adjacent nodes to propagate information, which is very effective for extracting graph representations. However, due to the high sensitivity of GCN to graph structure, different graph structures may lead to problems of over-smoothing or over-fitting. In webshell, the CFG graph features vary significantly between large, small, and one-word trojans, thus requiring improved model generalization ability.

MinCutPool is a graph clustering approach by solving a continuous relaxation of the normalized *minCUT* problem.The K-way normalized *minCUT* problem is the task of partitioning nodes $\mathcal{V}$ in K disjoint subsets (i.e., clusters) by removing the minimum volume of edges. In PHP source codes, the functions and parameters used to implement the same functionality exhibit continuity, making the corresponding block features similar in the CFG. These blocks will be called as functional clusters in the following text. We can aggregate them by optimizing this problem, which is equivalent to maximizing:

$$\frac{1}{K} \sum_{k=1}^{K} \frac{\sum_{i,j \in \mathcal{V}_k} E_{i,j}}{\sum_{i \in \mathcal{V}_k, j \in \mathcal{V} \setminus \mathcal{V}_k} E_{i,j}}, \tag{2}$$

where the numerator counts the number of edges in the same functional cluster and the denominator counts the number of edges between different functional cluster. Let $\mathbf{C} \in \{0,1\}^{N \times K}$ be a functional cluster assignment matrix, s.t. $\mathbf{C}_{i,j} = 1$ if node i belongs to functional cluster j. Then the *minCUT* problem can be expressed as:

$$maximize \frac{1}{K} \sum_{k=1}^{K} \frac{\mathbf{C}_k^T \mathbf{A} \mathbf{C}_k}{\mathbf{C}_k^T \mathbf{D} \mathbf{C}_k}, \tag{3}$$

where $\mathbf{C}_k$ is the k-th column of $\mathbf{C}$ and $\mathbf{D}$ is the degree matrix of $\mathbf{A}$. For a given input H^l, the MinCutPool layer uses a multi-layer perceptron (MLP) with the softmax activation function to predict the functional cluster assignments:

$$\mathbf{C} = MLP(H^l, \theta_{MLP}), \tag{4}$$

where θ_{MLP} is the parameters of MLP and optimized using the *minCUT* problem loss. According to the func-

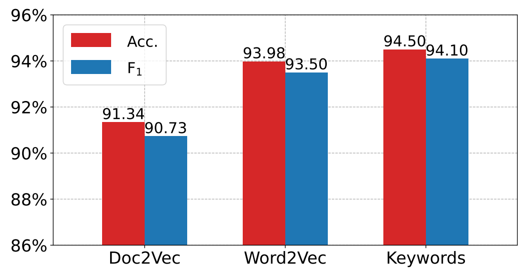


Fig. 3. The experimental results of different syntax extraction methods.

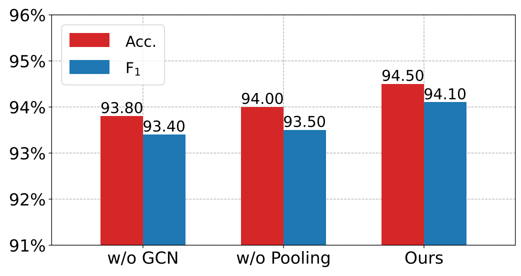


Fig. 4. Ablation study.

tional cluster assignment matrix $\mathbf{C}$, we perform pooling aggregation for the blocks in the same functional cluster.

IV. Experiments

A. Experimental Setup

In this section, we present and discuss our experimental results. Especially, we answer the following three research questions:

- **RQ1**: How effective is our graph representation for modeling PHP source code? (Section IV-B)
- **RQ2**: How well does our structural-aware model detect webshells? (Section IV-C)
- **RQ3**: If SAWD achieves better performance than state-of-the-art webshell detection systems? (Section IV-D)

1) Data Preparation: Currently, there is no standardized dataset available to evaluate the effectiveness of webshell detection systems. Thus, we manually collect webshells from 46 Github repositories and obtain normal samples from popular PHP projects with large user bases, such as Thinkphp and WordPress. We first remove duplicates from the collected webshells and normal samples via the md5 hash. Then, we classify the webshells according to our standards into three categories: big trojan (length greater than 2000 bytes), small trojan (length between 200 bytes and 2000 bytes), and one-word trojan (length less than 200 bytes). We obtain 911 big trojans, 1433 small trojans, and 407 one-word trojans. Finally, this collected webshell dataset is divided into a training dataset and a test dataset according to the proportion of 50% and 50%.

2) Experimental Settings: We train SAWD with an SGD optimizer for 200 epochs, and the learning rate is set to $5 \times e^{-4}$. The proposed approach is implemented using Python 3.8.0 and PyTorch 1.7.1.

B. Efficacy of Graph Representation (to RQ1)

To evaluate the efficacy of the graph representation for modeling PHP source code, we compare the performance of our keyword-based method with other syntax feature extraction methods. To implement the Word2Vec approach, we limit the feature extraction process to the first 40 tokens of each basic block. Besides, we assign a 150-dimensional vector for each token, resulting in a 6000-dimensional representation for each basic block. In the case of the Doc2Vec method, we extract features with a dimension of 1200 for each basic block, which is the same dimension as our keywords-based method. As Fig. 3 shows, the keywords-based method achieves an accuracy of 94.50% and outperforms Word2Vec and Doc2Vec. This benefits from its ability to selectively consider only the most informative words, thus retaining the original information of basic blocks to the maximum extent. In contrast, the Word2Vec method consumes excessive resources by storing all the features of each word in a block and must truncate the words, resulting in the loss of some key semantics. As for the Doc2Vec method, it considers the text of each block as a whole, which leads to the influence of many low-information words on feature extraction.

C. Detection Performance of Our Model (to RQ2)

To evaluate the detection performance of our structural-aware model, we generate AST from PHP source code and adopt CNN as the classifier. Note that we use the same top 1200 most frequent keywords as block features to ensure fairness. As shown in Fig. 4, we can observe that our method performs better than the CNN-based model. The reason is that our model can leverage GCN to combine graph representation for more efficient feature extraction. In addition, we can observe that the accuracy drops by about 0.5% after removing graph pooling, indicating that graph pooling indeed improves the generalization ability.

D. Comparison with Other Methods (to RQ3)

To demonstrate the effectiveness of our approach, we compare SAWD with a range of state-of-the-art webshell detection systems on our collected dataset. We use NeoPI [14] to extract multiple statistical features from the

Method	Accuracy	F_1 Score	Miss Rate
Statics+SVM	61.5%	69.4%	8.7%
Statics+RF	90.9%	90.0%	14.2%
Statics+MLP	87.3%	87.2%	11.8%
Opcode+SVM	92.0%	91.4%	10.2%
Opcode+RF	92.3%	91.8%	9.7%
Opcode+MLP	92.5%	91.9%	10.8%
SC+W2V+GRU	93.8%	93.2%	4.7%
MSDetector	93.3%	92.7%	6.3%
SAWD (Ours)	**94.5%**	**94.1%**	**3.4%**

TABLE I. Comparision of metrics with other methods.

source code files and use support vector machines (SVM), random forests (RF), and multi-layer perception (MLP) as the classifier. The opcode-based methods [15] adopt FastText to obtain the vectorized features of the opcode sequence, which is input into three machine learning algorithms for webshell detection. In addition, two DL-based methods proposed in [4] and [8] are also introduced for comparison.

We can observe from TABLE I that DL-based detection methods generally outperform traditional machine learning methods. This is because DL-based methods can automatically extract higher-level features from the data. Moreover, benefiting from SAWD preserving more complete syntax and semantic information, our method achieves better performance than other DL-based methods with an accuracy of 94.5%. Thus, it can be concluded that SAWD can effectively analyze PHP source code for webshell detection.

V. Conclusion

In this paper, we propose SAWD, a structural-aware webshell detection system, for PHP source code analysis. SAWD introduces an effective graph representation to preserve the syntax and semantic information in PHP source code. Based on the graph representation, SAWD leverages GCN combined with graph pooling for more efficient code analysis. To evaluate the performance of our proposed method, we conduct extensive experiments on the collected webshell dataset. The results demonstrate that our method outperforms state-of-the-art webshell detection systems. In future work, we will explore graph construction methods to replace CFG to further improve detection performance.

Acknowledgment

We would like to thank Robin Luo for his support and constructive comments on this work. This work was supported in part by SJTU-QI'ANXIN Joint Lab of Information System Security, NIO Student Innovation Funding Program, and Network Security & Ecosystem Innovation Laboratory of Intelligent Radio & Television Broadcasting of the National Radio and Television Administration under Grant TXX20200001ZSB001. Zhi Xue and Ruijie Zhao are the corresponding authors (e-mails: zxue, ruijiezhao}@sjtu.edu.cn).

References

[1] W. Yang, B. Sun, and B. Cui, "A webshell detection technology based on HTTP traffic analysis," in *12th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, vol. 773, 2018, pp. 336–342.

[2] L. Shi and Y. Fang, "Webshell detection method research based on web log," *Journal of Information Security Reserach*, vol. 2, no. 1, pp. 66–73, 2016.

[3] CiscoCXSecurity, "Neopi," https://github.com/CiscoCXSecurity/NeoPI.

[4] T. Li, C. Ren, Y. Fu, J. Xu, J. Guo, and X. Chen, "Webshell detection based on the word attention mechanism," *IEEE Access*, vol. 7, pp. 185 140–185 147, 2019.

[5] R. Zhao, S. Yong, H. Zhang, J. Long, and Z. Xue, "Webshell file detection method based on tf-idf," *Computer Science*, vol. 47, 2020.

[6] W. Kang, S. Zhong, K. Chen, J. Lai, and G. Xu, "Rf-adacost: Webshell detection method that combines statistical features and opcode," in *Frontiers in Cyber Security*, vol. 1286, 2020, pp. 667–682.

[7] Y. Li, J. Huang, A. A. Ikusan, M. Mitchell, J. Zhang, and R. Dai, "*ShellBreaker*: Automatically detecting php-based malicious web shells," *Comput. Secur.*, vol. 87, 2019.

[8] B. Cheng, Y. Guo, Y. Ren, G. Yang, and G. Xu, "Msdetector: A static PHP webshell detection system based on deep-learning," in *16th International Symposium on Theoretical Aspects of Software Engineering*, vol. 13299, 2022, pp. 155–172.

[9] T. W. Reps, "Program analysis via graph reachability," *Inf. Softw. Technol.*, vol. 40, no. 11-12, pp. 701–726, 1998.

[10] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *IEEE Symposium on Security and Privacy*, 2014, pp. 590–604.

[11] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, "Efficient and flexible discovery of PHP application vulnerabilities," in *IEEE European Symposium on Security and Privacy*, 2017, pp. 334–349.

[12] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *5th International Conference on Learning Representations*, 2017.

[13] F. M. Bianchi, D. Grattarola, and C. Alippi, "Mincut pooling in graph neural networks," *CoRR*, vol. abs/1907.00481, 2019.

[14] Y. Wu, M. Song, Y. Li, Y. Tian, E. Tong, W. Niu, B. Jia, H. Huang, Q. Li, and J. Liu, "Improving convolutional neural network-based webshell detection through reinforcement learning," in *23rd International Conference on Information and Communications Security*, vol. 12918, 2021, pp. 368–383.

[15] Y. Fang, Y. Qiu, L. Liu, and C. Huang, "Detecting webshell based on random forest with fasttext," in *International Conference on Computing and Artificial Intelligence*, 2018, pp. 52–56.