Hindawi
Security and Communication Networks
Volume 2021, Article ID 9923234, 13 pages
https://doi.org/10.1155/2021/9923234

Research Article

PBDT: Python Backdoor Detection Model Based on Combined Features

Yong Fang, Mingyu Xie, and Cheng Huang

School of Cyber Science and Engineering, Sichuan University, Chengdu, China

Correspondence should be addressed to Cheng Huang; opcodesec@gmail.com

Received 1 April 2021; Accepted 31 August 2021; Published 14 September 2021

Academic Editor: Shah Nazir

Application security is essential in today's highly development period. Backdoor is a means by which attackers can invade the system to achieve illegal purposes and damage users' rights. It has posed a serious threat to network security. Thus, it is urgent to take adequate measures to defend such attacks. Previous research work was mainly focused on numerous PHP webshells, with less research on Python backdoor files. Language differences make the method not entirely applicable. This paper proposes a Python backdoor detection model named PBDT based on combined features. The model summarizes the common functional modules and functions in the backdoor files and extracts the number of calls in the text to form sample features. What is more, we consider the text's statistical characteristics, including the information entropy, the longest string, etc., to identify the obfuscated Python code. Besides, the opcode sequence is used to represent code characteristics, such as TF-IDF vector and FastText classifier, to eliminate the influence of interference items. Finally, we introduce the Random Forest algorithm to build a classifier. Covering most types of backdoors, some samples are obfuscated, the model achieves an accuracy of 97.70%, and the TNR index is as high as 98.66%, showing a good classification performance in Python backdoor detection.

1. Introduction

With the rapid development of network technology in recent years, various applications have become the primary way to provide information services, significantly improving the convenience of people's life. However, once criminals utilize it, it will cause data leakage and property damage. Among them, the backdoor is an effective means for attackers to achieve the purpose of intrusion. The 2020 State of Malware Report [1] released by Malwarebytes Labs shows that backdoors have continuously become the top ten common security threat's categories among users and commercial products, and the proportion is increasing. Sufficient identification of backdoors to take further measures has become the current research's focus in cyber security.

As a concise, easy-to-read, and extensible programming language, Python has been widely used in large-scale project development while initially only writing automated scripts. Nevertheless, there is little research on malicious Python code. Most of the existing backdoor-related research is aimed at PHP or general-purpose webshells [2–8] and has not considered other backdoor types. Simultaneously, due to the differences in programming languages, existing methods are not fully applicable to Python text. There are feature selections for programming language functions or programming features in the research methods, such as PHP language tags "<?php. . .?>", "shell_exec ()" functions, etc. These features should have corresponding changes to the Python language. At the same time, the previous research did not involve a more detailed analysis and summary of function behavior of the backdoor. We think that these are essential when judging the maliciousness.

The attacker will use the backdoor to perform a series of subsequent operations, which must leave traces on the victim's host. Some studies [9, 10] use the dynamic acquisition of logs and other information to capture backdoor behaviors for detection. However, some can use system processes to hide their existence, such as thread insertion backdoor (Section 2.1), and this method does not identify well. As deep learning is widely used in the security field, some papers [11–14] use deep neural networks to check webshells and achieve good results. Nevertheless, Python

backdoor dataset is sparse, so training a deep neural network may cause overfitting and consume numerous system resources, resulting in poor overall performance. Moreover, the full-text feature extraction ignores the functions and modules that implement the backdoor's basic functions. Detailed research on malicious python code behavior has become a research point that needs to be broken.

This paper uses machine learning method for Python backdoor detection by combining multiple features such as *function calls*, *text statistics*, and *opcodes*. First, we analyze the modules and functions required for the basic functions of Python backdoor, including *text encryption*, *network communication*, *process settings*, *file operations*, *command execution*, and *system control*. Then, we count the number of times the suspicious module or function appears in the text and record the suspicious function's code line. What is more, the entire text's statistical characteristics, including *information entropy*, *longest string*, *coincidence index*, and *compression rate*, are obtained to capture the characteristics of obfuscated codes. We also consider the number of IP, URL, and many dangerous keywords that frequently appear in the backdoor. In addition, using the opcode information, the code is represented by the statistical features of the overall text and suspicious function lines, the TF-IDF feature representing the contextual relevance, and the FastText feature. Finally, it is sent to the Random Forest classifier to determine whether it belongs to the backdoor. The main contributions of this paper are as follows:

(1) In response to lack of dataset, we manually generated some Python backdoor samples, including rebound shell and obfuscated code, to expand the sample library and improve the accuracy of classifier detection.

(2) By analyzing many real samples, the malicious functions and modules frequently used in the Python backdoor are summarized to distinguish the benign and malicious sample. We integrate multiple types of features to classify malicious codes. Both statistical features and opcode features can eliminate the influence of obfuscated codes and ignore irrelevant information such as text comments. Simultaneously, the n-gram value selection fully considers the Python opcode's characteristics and accurately represents the text information.

(3) We collect a total of 2,026 samples to test the proposed model, including 1,511 benign Python codes and 515 Python backdoor codes. In the end, all indicators of PBDT reach more than 95%, and the accuracy is as high as 97.7%.

To verify the validity of PBDT, a series of comparative experiments are designed. The results show that, compared with other machine learning algorithms, Random Forest has better performance in the classification of Python backdoors, and the performance of the combined feature is better than any single feature, which also has a more remarkable performance improvement than the previous paper method.

The rest of this paper is organized as follows. Section 2 introduces the relevant background, including the definition and classification of backdoors and previous research work. We will describe the PBDT system architecture and specific implementation methods in Section 3. Subsequently, in Section 4, the dataset and the evaluation results of PBDT are explained and analyzed. Finally, Section 5 summarizes the full study.

2. Background

2.1. Backdoor. Thomas and Francillon [15] definition of the backdoor is that the intentional structure existing in the system undermines the original security of the system by providing convenient access to other privileged functions or information. In other words, the backdoor refers to a program method that bypasses security controls such as authentication to obtain system permissions, whose common types include webshell, C/S backdoor, thread insertion backdoor, and extended backdoor:

(1) Webshell refers to the backdoor program that exists in the web application. The file format includes PHP, ASP, JSP, Python, and so on. After the attacker exploits the vulnerability to invade the website, the webshell is placed in the server file directory for subsequent remote control, execution of malicious commands, and other operations.

(2) C/S backdoor uses a client/server model to achieve the control operation, some of which are similar to traditional Trojan programs' principles. After the attacker implants the server into the target computer, the client starts the backdoor to control it. Rebound shell is also based on this mode, but the two roles are precisely the opposite. The attacker runs a server program to monitor a specific TCP/UDP port, and the victim host initiates a request for an active connection. This rebound connection is usually applicable to scenarios such as restricted firewalls, occupied ports, and insufficient permissions on the controlled end.

(3) Thread insertion backdoor does not have an independent process when it runs but inserts into a particular service or thread of the system, which is now a mainstream type of backdoor. However, it is relatively difficult to detect or kill, and traditional firewalls cannot effectively defend against it.

(4) The extended backdoor usually concentrates a variety of common functions, including system user detection, port opening, opening/stopping services, HTTP access, and file upload/download. It has powerful functions but relatively inadequate concealment.

Python has simple structures but powerful functions. Attackers only need to write scripts' dozens of lines to establish a persistent backdoor. And because Python is a common language used by administrators, there is no noticeable difference between malicious Python traffic and the

traffic generated by daily network management tools, so it is difficult to be detected by the terminal detection response system, and it is quite popular with hackers. As a result, the traditional backdoor recognition method is not universal, and it is particularly important to realize an effective detection of Python backdoor scripts.

2.2. Related Work. At present, there is little research on Python malicious code in security field. The papers on backdoor detection mainly focus on webshell. At the same time, JavaScript is also used as an interpretable language, and part of the detection methods of malicious statements can be used in other language types. This section will introduce existing research on webshell and malicious JavaScript from the perspective of static detection and dynamic detection. Among them, the more typical papers are summarized in Table 1.

2.2.1. Static Detection. Static detection mainly identifies malicious code by analyzing the grammatical structure and statistical characteristics of the source code. The most classic method is to build a black or white list and detect malicious files through simple string's regular matching. NeoPi [16] is a classic webshell open-source detection tool. It considers the document's statistical characteristics to determine whether it is obfuscated and matched some feature function for detection. However, the feature database is relatively old. Tu et al. [20] detected webshells in applications based on malicious signatures and malicious functions. At the same time, it was proposed to consider only the longest word's beginning and ending with the header tag. This method reduced the false positives from NeoPi's 24.5% to 6.7%, but its detection essence was still simple character matching. Lawrence et al. [2] designed a firewall tool to intercept and alert calls to system functions that were not in the whitelist. However, due to the limited whitelist, there were high false positives, and the webshell that was encrypted and obfuscated by complicated means could not be identified.

It is impossible to obtain the function's language environment only by manually defined black and white lists, which will cause many false positives. Besides, due to the continuous variation of malicious code types, the rule base's update is critical, and false negatives are inevitable. With the widespread application of machine learning in various fields, it has also been used to detect malicious code, improving detection accuracy by combining with information such as code syntax and semantics. AL-Taharwa et al. [21] proposed and implemented a JavaScript obfuscation detector JSOD, which focused on obfuscated scripts. It first performed anti-obfuscation processing and extracted the contextual semantic information of the code by using the AST (Abstract Syntax Tree). Then the malicious JavaScript was detected by the Bayesian classifier. Fass et al. [17] proposed a pre-detection system JStap for malicious JavaScript. Based on existing lexical and AST detection, the system added the code CFG (Control Flow Graph) and PDG (Program Dependency Graph), fully considering the syntax and semantic information of the program. Finally, it was classified by Random Forest. While using machine learning, the effect of basic functions on the maliciousness of the code cannot be ignored.

The most common language type in webshell is PHP. Opcodes are the instructions and fields of the PHP operation performing, which can eliminate interference from irrelevant items. In recent years, it has been often used in PHP webshell detection. Cui [18] designed a webshell detection model, RF-GBDT, which considered the statistical characteristics of PHP files. Furthermore, the model extracted the TF-IDF vector and the hash vector of the opcode sequence. After integrating all the features, the webshell was recognized through Random Forest and gradient boosting decision tree. Fang et al. [22] proposed and implemented the PHP webshell detection system FRF-WD. They innovatively used the FastText text classifier to characterize PHP's opcode sequence, integrating its classification results and statistical features as the Random Forest classifier's input. In addition to the statistical and opcode features mentioned above, Pan et al.'s [23] detection method of webshell used AST to obtain executable data features of PHP code, fully considering the execution data flow and function parameter characteristics of common system commands. The opcode sequence should be combined with the best n-gram value to effectively ensure the detection effect.

With the rapid development and application [24, 25] of deep neural networks, opcodes are often combined to realize the webshell detection function. Yong et al. [14] detected PHP webshell based on DNN (deep neural network), extracted opcode features through 2-gram and TF-IDF grammar models, and input them into a DNN model composed of CNN (convolutional neural network), LSTM (long short-term memory), and MLP (multilayer perceptron) for detection. But the model consumes a lot of computing and storage resources. Word vector is a commonly used representation method in text classification, and it has also shown good detection advantages in malicious code detection in the recent years. Tian et al. [12] detected webshells with HTTP request packets, used Word2vec to convert the collected packet text into word vectors, and finally implemented classification through the CNN model. Ndichu et al. [26] proposed a malicious JavaScript detection scheme, which used the Doc2vec model to characterize the source code context's features and input them into the SVM classifier to determine the program's maliciousness.

In summary, static detection can use limited resources to achieve better detection results, but false negatives and false positives are inevitable. How to improve detection accuracy is a problem that needs to be focused on. The relevant research introduced above is aimed at languages such as PHP and JavaScript. Improving the existing methods to make them suitable for Python malicious code detection is a direction worth studying.

2.2.2. Dynamic Detection. The main idea of dynamic detection is to dynamically execute sample files, monitor network traffic, or call sensitive functions to identify malicious code. This method is often used to analyze specific

Table 1: Summary of related work.

	Author	Main work	Shortcoming
Static detection	Scott and Hagen [16]	Identifying obfuscated webshells through statistical features	The feature library is old, and using simple character matching with high false positives
	Fass et al. [17]	Extracting JavaScript semantic information through AST, CFG, and PDG for malicious judgment	No analysis and consideration of basic functions with malicious intent
	Cui et al. [18]	Using TF-IDF vectors and hash vectors to obtain webshell opcode features for detection	No semantic information is considered, which may result in false negative
	Yong et al. [14]	Processing opcode through 2-gram and TF-IDF, and using composite neural network DNN for webshell's classification	Deep neural network is too complex and consumes a lot of resources
Dynamic detection	Canali and Balzarotti [9]	Analysis of common webshell behavior using honeypot technology	High requirements for resources, environment, and samples
	Wang et al. [19]	Combining attack feature's vectors and dynamic execution trajectories	The types of malicious functions summarized are not comprehensive

behavior types of files, including extracting HTTP requesting or responding to payload characteristics, and hooking sensitive functions.

Kim et al. [27] designed a framework called JsSandbox to detect malicious JavaScript. Through IFH (internal function hooking) monitoring and analysis of sample code behavior, functions that could not be executed by API hooking could be extracted. This method was commonly used in other malicious code classification tasks and would not be affected by operations such as code deformation and obfuscation on detection performance. Canali et al. [9] used honeypot technology to obtain and analyze the attacker's behavioral characteristics' destruction of the target. The information source included HTTP request logs and files modified or generated after the attacker obtained the victim host's permission. It also focused on analyzing common webshell behaviors. Xie and Hu [10] developed an anomaly detection system that can identify webshells. They analyzed ADFA-LD intrusion detection datasets, obtained log behaviors, and used the K-nearest neighbor (KNN) algorithm to cluster them. Although dynamic detection can effectively reduce the rate of false positive and false negative, it consumes more resources and it is not suitable for detection tasks with multiple samples. At the same time, there may be a big gap between the actual application and the detection effect in the ideal experimental environment.

Therefore, studies have combined static and dynamic detection to achieve better classification results with limited resources. Rieck et al. [28] proposed Cujo, a malicious JavaScript automated detection system, which analyzed by combining static lexical features and dynamic runtime features. The dynamic analysis used sandbox to obtain web page code, provided analysis reports, and finally mapped the report results to vector space. The final detection model was generated based on SVM machine learning. Wang et al. [19] researched and implemented the JavaScript malware detection tool JSDC. First, it used the extracted text, program information, and dangerous function call features to detect based on machine learning. According to attack feature vector and dynamic execution trajectory, malware was divided into eight attack types. The dynamic and static combination achieved a false positive rate of 0.2123% and a false negative rate of 0.8492%. Starov et al. [3] conducted an in-depth analysis of common webshell behaviors based on two dimensions of static analysis and dynamic analysis. Through static analysis, it was found that most of the sample attacks included file browsing and uploading, system information viewing, command execution, etc., while dynamic analysis found that most of them would try to access links or directories such as http://∗ and /etc/∗.

Dynamic detection performs better in identifying malicious code behavior, but its inherent defects have high requirements for resource environment and samples, and its application in practice is limited. It is necessary to consider various factors to select a suitable detection method comprehensively.

3. Methodology

The PBDT model proposed in this paper is constructed based on multiple features. Various features can be used for classification alone or in combination. Experimental data will be used to illustrate the performance of each combination in Section 4.3. The architectural designed is shown in Figure 1. It has three categories, including sample module and function *call features*, *text statistical features*, and *opcode features*.

The calling feature involves six types of modules and functions, including *text encryption, network communication, process settings, file operations, command execution*, and *system control*, to capture potentially dangerous behaviors of the sample. However, the backdoor file may be obfuscated to avoid detection, so text statistical features are added to identify the obfuscated sample file through typical parameters such as *information entropy, longest string, coincidence index*, and *compression rate*. At the same time, this part involves the IP, URL, and dangerous keywords that are likely to exist in the backdoor. It is impossible to identify the backdoor based on the above-mentioned manually defined malicious behaviors accurately. Therefore, the opcode-related features are added, including the simple statistical features of the overall text and the suspicious function line, the TF-IDF feature indicating the importance of words, and the FastText feature of the efficient text classifier to comprehensively analyze the meaning of the opcode. Finally, these features are combined as the feature vector and input

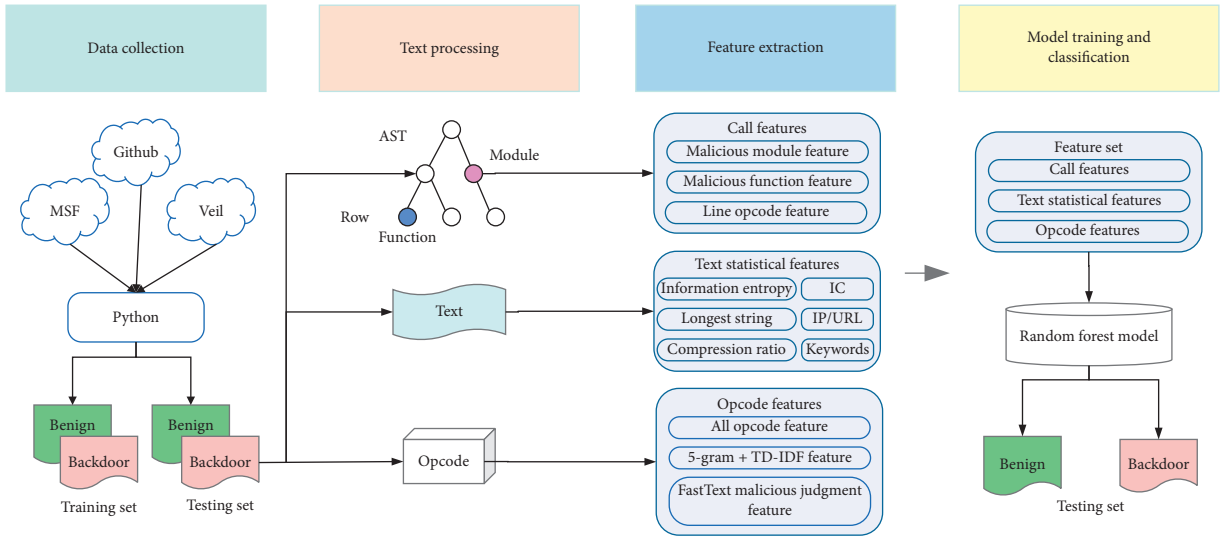

Figure 1: Architecture of PBDT.

into the Random Forest classifier to classify Python samples and accurately distinguish backdoor files with malicious behavior.

Some of the above features are proposed by previous researchers and some are proposed in this paper, which we summarize and illustrate in Table 2. For the old features, the improvements made in this paper are noted in the third column of the table. In the following, various features and classifier selection will be introduced in detail.

3.1. Call Features. It can be seen from the definition of backdoor that the two key points worth noting are "bypassing security controls" and "obtaining system permissions." Therefore, there must be specific modules and functions to implement corresponding functions. We collect and analyze numerous backdoor samples, combining with data summary and empirical analysis, and divide the backdoor's common behaviors into the following six categories (Sections 3.3.1–3.1.6). At the same time, we list the modules and functions required to implement each type of function. See Table 3 for details.

Most of the function's acquisition in the file directly performs regular matching on the text, such as[16]. The shortcomings of this method are obvious. When the corresponding function exists in the comment, it will be regarded as called. But in fact, the function has not been executed, and of course, there will be no malicious behavior. This paper analyzes the AST of Python code, extracts the file import module and call function information, and obtains the number of various dangerous modules and functions after counting, as the call features part of the final classification.

AST is a tree-like representation of the abstract grammatical structure of a programming language. As the input of the compiler's back-end, it does not depend on specific grammar and language details and represents information on the semantic level of code. Currently, AST has been widely used in the detection of malicious code [17, 21, 29–33]. This method can eliminate the influence of annotations on text analysis and effectively extracts information about imported modules and calling functions. If it is determined to be a suspicious function, recording the number of lines in the source file to obtain the line's opcode information (Section 3.3.1) and finally returns it as a result for in-depth study and analysis of suspicious file behavior.

3.1.1. Text Encryption. The backdoor uses some obfuscation methods to bypass the security control and evade the software's detection and killing of system. The most common one is to use algorithms to encrypt text. In order to achieve this purpose, it is necessary to introduce corresponding functional modules and call functions, including commonly used encryption algorithms "AES," "RSA," "base64," encryption padding "padding," "OAEP," encoding conversion "binascii," etc., which can be used as determining a feature of the backdoor file.

3.1.2. Network Communication. A crucial function in the backdoor is data interaction, including the communication between server and client in the C/S backdoor, and webshell uploading or downloading files from a specific network address. Python language implements such functions including network communication "socket," "setsockopt," SSH connection to remote server "paramiko," "SSHClient," HTTP request "urllib," "httplib," etc. If there is such behavior, pay attention.

3.1.3. Process Setting. Generally, the process needs to be set when the backdoor is running, such as starting a new process to facilitate subsequent operations, including creating a process "subprocess," multiprocess management "multiprocessing," and generating a pseudo-terminal "pty." Concurrently, to avoid detection and obtain permissions, the process may monitor "select," modify, and obtain the

Table 2: Feature summary.

Feature set	Old features	New features and improvements
Call features		Malicious module feature Malicious function feature Line opcode feature
Text statistical features	Information entropy The longest string Index of coincidence Compression ratio	
		IP/URL information Dangerous keywords
Opcode features		All opcode features
	TF-IDF feature	5-gram
	FastText feature	5-gram

Table 3: List of common modules and functions of backdoor.

	Module	Function
Text encryption	AES/base64/binascii/hashlib/RSA/Crypto/sha256/hashes/padding/DES	b64encode/b64decode/encrypt/decrypt/EncodeAES/DecodeAES/AESGCM/md5/rc4/SHA256/sha1/encode_base64/OAEP/MGF1
Network communication	Socket/urllib2/urllib/paramiko/ftplib/SocketServer/httplib	Socket/bind/setsockopt/gethostbyname/gethostname/SSHClient
Process setting	Subprocess/commands/pty/threading/select/multiprocessing/setproctitle	spawn/Popen/communicate/daemon/fork/ThreadingTCPServer/ThreadingUDPServer/setproctitle/CreateThread
File operation	Shutil/fcntl/StringIO/BytesIO/ctypes/scapy.all	Exec/execv/execvp/execfile/storbinary
Command execution	Argparse/getopt/getpass/argv/optparse/cmd	System/getopt/getoutput/tcsetattr/command/exec_command/check_output
System control	Platform/winreg/psutil/wmi/pynput	VirtualAlloc/sysinfo

process name "setproctitle", etc. The introduction of above module has reason to suspect that it has an attacking intent.

3.1.4. File Operation. The backdoor will also perform a series of operations on files, including polluting local files, reading and writing system sensitive files, and replacing common system command files, in order to achieve the purpose of obtaining system permissions. For example, memory read and write "StringIO," file copy "shutil," file lock "fcntl," and "exec" family functions for executing files, including "execl," "execv," and "execvp." Although the above modules and functions are harmless by themselves, they may cause undesirable consequences when used by the backdoor.

3.1.5. Command Execution. Executing system commands is a common and potentially harmful behavior of backdoors and may be a pivotal step in achieving core functions. Specifically, it includes the functions "system," "command," "exec_command," etc., that execute commands. In addition, interacting with the command line and parsing parameters is also a significant feature, such as modules "argparse," "getopt," "argv," etc. But normal Python programs may also have such requirements, so the correct distinction is necessary.

3.1.6. System Control. Controlling the system and obtaining system-related information is the ultimate goal of backdoor execution and an essential manifestation of backdoor hazards. Such functions include system monitoring "psutil," system hardware's information acquisition "wmi," system operation "platform," etc. Besides, registry operations "winreg" and virtual memory allocation "VirtualAlloc" are also common behaviors. This part also includes some special functions, such as the module "pynput" that controls the keyboard and mouse. The above can be used as a vital basis for judging whether it is a backdoor.

3.2. Text Statistical Features. The matching of modules and functions can simply and intuitively identify some malicious behaviors of the backdoor. However, in actual applications, in order to bypass the security control, the backdoor file is likely to undergo obfuscation, such as the splicing or encoding of characters, and simple function matching cannot achieve the detection effect. The obfuscated code text has some typical features. Here, four types are selected as the final feature components, and two additional features are added simultaneously.

3.2.1. Information Entropy. The concept of information entropy was first proposed by Shannon [34] in 1948. It represents the probability of random discrete events and is a measure of the system order's degree. The more chaotic the information, the higher the corresponding entropy. The method of calculating information entropy is as follows:

$$H(x) = -\sum_{i=1}^{n} p(x_i)\log(p(x_i)). \quad (1)$$

Among them, $p(x_i)$ represents the probability of a random event x_i. When the backdoor performs file obfuscation to hide its existence, encryption and encoding will generate some random strings, making the information entropy higher. Therefore, the greater the information entropy, the more suspicious the corresponding file.

3.2.2. The Longest String. When the code is obfuscated and encoded, it will generate a long string without spaces, such as the classic base64 encoding. Moreover, the introduction of shellcode will have the same effect, which is not easy to detect because of the malicious behavior hidden in the hexadecimal machine code. This kind of long string is rarely encountered in regular Python code, so the length of the longest string in the code can be used to determine the backdoor.

3.2.3. Index of Coincidence. Index of coincidence represents the relative frequency of the letters in sample, that is, the probability that two letters are the same. The concept is widely used in cryptography related research. The calculation method is as follows:

$$\mathrm{IC} = \frac{\sum_{i=1}^{c} n_i(n_i - 1)}{N((N-1)/c)}. \quad (2)$$

C is the normalization coefficient, which is 26 for English letters, n_i is the number of times each letter appears in the text, and N represents its length. Since the encrypted text's randomness will increase, the code has a high probability of being a backdoor file after obfuscation operations such as encryption and encoding if the IC value is low.

3.2.4. Compression Ratio. In information theory, data compression is a process of representing information with fewer data bits than the source file according to a specific encoding mechanism, which can make the distribution of data characters tend to be balanced. The ratio of after compressing to before compressing of a file's size is called the compression ratio. In order to achieve the purpose of obfuscation, malicious files generally have a relatively uniform character's distribution after particular encoding, and the data compression rate is higher than that of ordinary files. It is reasonable to think that files with high compression rate are backdoors.

3.2.5. IP/URL Information. When the backdoor performs network communication, IP or URL is most likely required to indicate the target address, which usually appears in pairs with data interaction related functions. For example, "bind" binds the IP and port, "host" parameter indicates the communication host, and "connect" connects to the IP address and also includes interaction with a specific network address URL. Therefore, the total number of IPs or URLs in the code can be used as a feature of the backdoor.

3.2.6. Dangerous Keywords. When developers write backdoor code, due to their habits or functional needs, they will include some backdoor-related words in comments or named functions. The characters considered here include "shellcode," "webshell," "shell," "backdoor," "cmd," "command," "hack," and "bypass." If there are more dangerous keywords in the code, it is considered as a suspicious backdoor file.

3.3. Opcode Features. The two modules mentioned above can only identify known malicious behaviors or encoded special files. However, there are various types of backdoors in actual applications. People have limited awareness of malicious codes, and not all malicious files will be encoded. The text's characteristics of some encoded files are not prominent. Therefore, we consider adding opcode features to express Python code through the nature of instructions and contextual connections represented by the opcode itself.

Opcodes represent instructions or fields for performing operations in a computer program. It is a step in compilation process of Python files. Compared with the source code, it can eliminate the influence of file obfuscated and ignore the interference items such as source comments. There are 121 opcodes listed in the latest Python3.8 document. This paper obtains the sample's opcode sequence and extracts the corresponding features through the following three types of processing.

3.3.1. Statistical Features. The functions and processing methods of backdoor files are different from benign files, so the types and numbers of opcodes used may be different. First, we conduct statistics on each opcode instruction for the entire text and generate an array with a length of 121 as the overall opcode statistical feature.

The primary manifestation of the malicious function performed by the backdoor is malicious functions. If there are suspicious functions in the benign file, the opcodes of the two contexts' execution parameters may be different. Therefore, the backdoor file can be identified by the opcode characteristics of the suspicious function line. Use the lines' number of the code file's dangerous function when obtaining function information through AST. Then, compile all code lines containing dangerous functions into opcodes separately, and count the total number of various opcodes. Similarly, an array of length 121 is generated as the statistical feature of the malicious function line's opcode.

3.3.2. TF-IDF Feature. TF-IDF (term frequency-inverse document frequency) is a commonly used weighting technique to evaluate the word's importance to the entire text [35]. TF represents the frequency of target word in the text. IDF decreases as the number of texts containing a certain phrase increases. It is used to reduce the impact of some common phrases in all texts that have little effect on the function. The two are multiplied to the final TF-IDF value. The calculation formula is as follows:

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}},$$

$$idf_i = \lg \frac{|D|}{\left|\left\{j: t_i \in d_j\right\}\right|}, \quad (3)$$

$$tfidf_{i,j} = tf_{i,j} \times idf_i.$$

Among them, $n_{i,j}$ refers to the number of occurrences of the word i in file j, and the corresponding denominator represents the sum of occurrences of all words in file j. $|D|$ refers to the total number of samples, and the corresponding denominator indicates the number of documents containing the term i. If the term is not in the document, the denominator will be zero. Generally, the denominator will be increased by one in practical applications. The main idea applied here is that if a certain phrase appears more frequently in a given sample and rarely appears in other samples, it is considered that the phrase has a greater class distinction ability.

In order to obtain the local context information of the text, on the basis of obtaining the full-text opcode sequence, the n-gram grammar model is used to segment the sequence and calculate the frequency. The basic idea is to perform a sliding window operation of size n on the text content in bytes to form a sequence of byte fragments with a window of n. Here, $n = 5$ is used to balance the completeness of the information expressed by opcode and the effect of model (Section 4.3.1). The first 50 subsequences are selected for calculation since the amount of information and model performance is comprehensively measured. The word sequence's information of the backdoor and benign files may be different, so the TF-IDF value is calculated according to the n-gram segmentation result. The opcode feature matrix indicating the importance of the phrase is obtained to identify the backdoor.

3.3.3. FastText Malicious Judgment Features. FastText is a word vector representation and fast text classification tool open-sourced by Facebook in 2016 [36]. It provides a simple and efficient method for text classification and representation learning. It can achieve accuracy comparable to deep learning methods but is many orders of magnitude faster than its training speed. The length of the opcode sequence of different files may be quite inconsistent. The fixed-length word vector's embedding may miss the text's critical information or do much useless work that consumes multiple computing resources. Using the FastText model's prediction result as a feature is more suitable for this type of data.

FastText has two important optimizations, n-gram and hierarchical softmax. The model architecture is shown in Figure 2, where $x_1 - x_N$ represent the n-gram vector in the text, and the average value of word vector is used as the feature to predict the specified category. The value of n selected here is the same as the previous module; both are 5. N-gram will keep word order information during model training. The softmax function is often used as an activation function in the neural network's output layer to normalize the output value of the neuron to 0-1 interval. Compared with the standard softmax, which normalizes all categories' probabilities, hierarchical softmax constructs a Huffman tree based on the probability of the category, which can reduce the complexity from N to logN and significantly improve the efficiency of model. The label category of test set predicted by the FastText model is used as a feature of the backdoor's judgment.

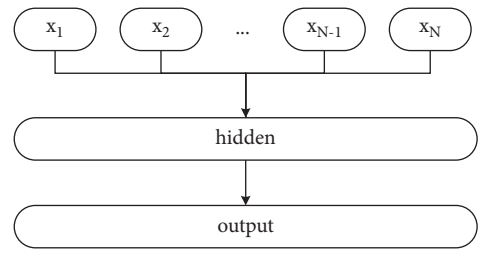

Figure 2: FastText model architecture.

3.4. Classifier. After integrating all the above features, they are sent to the classifier for model training and sample detection. Here, a Random Forest classifier is selected. This classifier was first proposed by Tin Kam Ho of Bell Laboratories in 1995 [37] and has been widely used since then. Random Forest is a classifier containing multiple decision trees, and the mode of output category of a single tree is used as the final output. It uses unbiased estimation. The model has a robust generalization ability, and it is insensitive to missing or outlier values of features. For unbalanced datasets, it can reduce errors.

This paper selects several classic machine learning models, including XGBoost, Naive Bayes (NB), and Support Vector Machine (SVM). Compared with the traditional GBDT algorithm, XGBoost supports column sampling, which can not only reduce overfitting, but also reduce calculations. Naive Bayes performs well on small-scale data and has a high speed during training, while SVM has excellent generalization ability. The above classifiers perform well in a variety of code classification tasks. We conduct performance comparisons through experiments (Section 4.3.3) and find that the Random Forest has the best classification effect, so we believe this algorithm is suitable for Python backdoor detection.

4. Experimental Evaluation

This section will evaluate the effect of PBDT and prove its benefits through comparative experiments. First, we used tool generation and network collection methods to obtain more than 2,000 Python samples. After deleting some low-quality data, we labeled them. Then, we built a model, used existing data to evaluate its performance, and compared the detection effects of various modules and common algorithms. Finally, we compared the performance of PBDT with some previous detection methods in similar fields on this dataset. Experiments show that PBDT has a better distinguishing ability for Python backdoors.

4.1. Dataset. The current research on malicious Python code is limited, and there is no relatively comprehensive and authoritative public dataset. We have extensively collected various Python files from the open-source library GitHub, including malicious backdoors and benign codes. Together, since there are few public backdoor samples, we also used some tools to generate them. In the end, 1,511 white samples and 515 black samples were obtained. In each experiment, they are randomly divided into 70% training set and 30% testing set to avoid the fortuity of results. The primary data sources are shown in Table 4.

4.1.1. Backdoor Data. The backdoor samples are mainly divided into three categories:

(1) First of all, we collected a wide range of GitHub projects, including webshells, reverse shells, C/S backdoors written in Python language, and so on. The collected files were marked in the project introduction and verified by manual inspection to be malicious.

(2) Another part of the samples are generated using Metasploit Framework (MSF), an open-source security vulnerability detection tool with functions including the whole penetration testing process. The msfvenom module can generate Trojan programs. We have obtained part of the rebound shell through this tool, including some samples encoded by base64 or containing shellcode.

(3) In actual applications, backdoors are mostly obfuscated. In order to obtain more comprehensive data, we use the Veil-Evasion anti-virus tool, which can be used to generate Metasploit payloads and bypass standard software detection or killing. Combining these two tools, a high-quality sample that can bypass security controls is obtained.

4.1.2. Benign Data. The benign samples are all obtained from GitHub, including the Python code of various basic functions. To ensure the data's accuracy, we try to choose the project with more stars, which shows that the project has a high degree of recognition. The standard code with ordinary communication function has some functional similarities to the backdoor, which may cause false positive in practical applications. Therefore, some samples like this are added to the benign dataset to ensure the accuracy of final training model's classification.

4.2. Evaluation Index. Choosing appropriate evaluation indicators is the key to the experiment. K-fold cross-validation is a conventional method for model evaluation. Generally, within a specific range, the evaluation accuracy will increase with the K value increase. However, the number of datasets in this paper is limited. When the K value is considerable, the test samples are small so that the result's validity cannot be guaranteed. Therefore, for each type of experiment described below, the dataset is randomly divided into a training set and testing set at a ratio of 7 : 3 and repeated ten times at the same time to calculate the average value of each indicator. This avoids the fortuity of the experimental results and reduces the number of samples' impact.

Table 4: Statistics from the main sources of data.

Type	Source
Benign	https://github.com/TheAlgorithms/Python https://github.com/Lawouach/WebSocket-for-python https://github.com/facert/socket-example https://github.com/geekcomputers/Python
Backdoor	Metasploit generation Veil-Evasion + Metasploit generation https://github.com/xl7dev/WebShell/tree/master/python https://github.com/JustinTom/Packet-Sniffing-backdoor

This paper is a typical two-category problem. The sample will be divided into positive and negative, and there are four possible results, as shown in the confusion matrix in Table 5. The horizontal direction is predicted category, and the vertical direction is actual category. In the matrix, TP (true positive) represents the number of correctly classified Python backdoors. FP (false positive) represents the number of benign samples that are mistaken as backdoors. TN (true negative) represents the number of benign samples that are correctly classified. Furthermore, FN (false negative) indicates the number of backdoor samples that are mistaken as non-malicious. Based on the confusion matrix, this paper uses the following evaluation indicators:

$$\begin{aligned} \text{accuracy} &= \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}, \\ \text{precision} &= \frac{\text{TP}}{\text{TP} + \text{FP}}, \\ \text{recall} &= \frac{\text{TP}}{\text{TP} + \text{FN}}, \\ \text{TNR} &= \frac{\text{TN}}{\text{FP} + \text{TN}}, \\ F_1 &= \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}. \end{aligned} \tag{4}$$

In the above indicators, the accuracy represents the proportion of correctly classified samples in all samples. The precision represents the probability that the samples predicted to be backdoors are malicious, while the recall represents the proportion of the actual backdoor samples that are correctly predicted (also called TPR, true positive rate). TNR (true negative rate) represents the ratio of correct predictions in actual benign samples, and F1 is the average index of precision and recall. In addition to the above five values, we also visually compare each algorithm's performance by drawing ROC (receiver operating characteristic) curves (Section 4.3.2).

Table 5: Confusion matrix.

	Predict backdoor	Predict benign
Actual backdoor	TP	FN
Actual benign	FP	TN

4.3. Basic Experiment. First, we make experimental selections on the value of n during the two n-gram processing in Section 3.3. At the same time, in order to verify the effectiveness of each feature, we select various features individually or remove them from the full features to evaluate the capability of the model. In terms of classifier selection, we compare the commonly used machine learning algorithms with the Random Forest used in this paper to ensure that the optimal algorithm is used. In this section's experiments, under the premise of comprehensively considering model efficiency and detection accuracy, when training the FastText model, the parameter word vector dimension is selected as 30. Meanwhile, the epochs value is 300, and the number of decision trees in the Random Forest is set to 100.

4.3.1. Reasonability of n in N-gram. Both FastText and TF-IDF in the feature involve the value of n in n-gram. We design experiments to verify the rationality of n. When n takes different values, a line graph is drawn for the model's accuracy and recall. The results are shown in Figure 3.

It is found that both indicators maintain a relatively high level when $n = 5$, indicating that the value of n in the feature is reasonable. When it is lower, it cannot accurately represent the complete sentence information of the Python opcode. When it is higher, the opcode sequence appears too few times in the code, so $n = 5$ can better represent the Python opcode's relevant features.

4.3.2. Feature Validity. All features are divided into three categories here. The text's statistical features in Section 3.3.2 are relatively simple and cannot accurately represent the sample, so the FastText feature in Section 3.3.3 is added to this category. Since the opcode feature of the malicious function line in Section 3.3.3 is based on the suspicious function defined in Section 3.3.1, it is included in the call feature. Three types of call features, text statistics and FastText features, and full-text opcode features are separately sent to the Random Forest classifier. We simultaneously combine them for experiments and compare them with all features. The experimental results are shown in Table 6.

It can be seen that various features have certain classification effects, but their performance is limited. After removing any type of features, the classifier's various indicators have declined, indicating that each feature plays an indispensable role in training the final model. The hybrid features proposed in this paper can better identify the Python backdoor, with an accuracy of 97.70% and a TNR of 98.66%.

4.3.3. Classifier Effectiveness. In order to verify the effectiveness of the algorithm, some classic machine learning algorithms that are widely used in research are selected, and

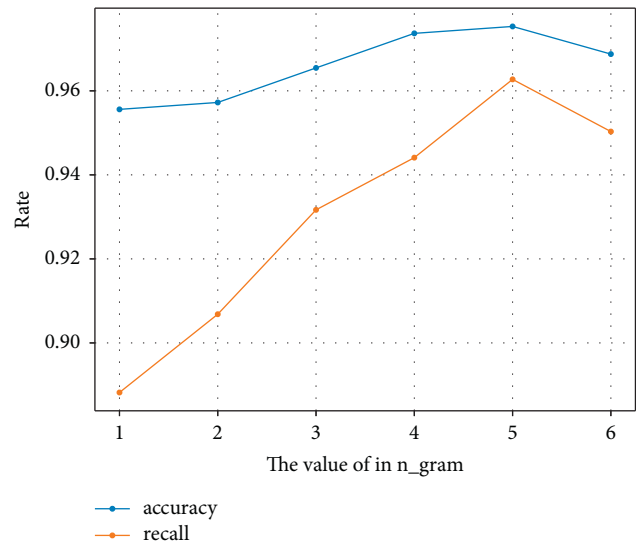

Figure 3: Corresponding indicators for different values of n.

Table 6: Performance comparison of different feature combinations.

	Features	Accuracy	Precision	Recall (TPR)	TNR	F1-score
#1	Call	0.9342	0.8805	0.8696	0.9575	0.8750
#2	Text	0.9391	0.9026	0.8634	0.9664	0.8825
#3	Opcode	0.9441	0.8757	0.9193	0.9530	0.8970
	#1+#2	0.9589	0.9146	0.9317	0.9687	0.9231
	#1+#3	0.9638	0.9264	0.9379	0.9732	0.9321
	#2+#3	0.9473	0.8817	0.9255	0.9553	0.9030
PBDT	#1+#2+#3	0.9770	0.9623	0.9503	0.9866	0.9563

the comprehensive feature's training model is used to compare the performance with the Random Forest classifier. The algorithms considered here include XGBoost, Naive Bayes, and Support Vector Machine. In order to compare the effects of each model more intuitively, the ROC curve of each algorithm during the test is drawn, as shown in Figure 4.

The algorithm's effect can be judged by the graph area's size formed by the ROC curve and the x-axis. The graph shows that the Random Forest has the best effect, followed by XGBoost, Naive Bayes, and Support Vector Machine. Therefore, it can be considered that the Random Forest classifier has the best applicability in Python backdoor detection.

4.4. Comparative Experiment. To further illustrate the advantages of PBDT in detecting Python backdoors, several representative models are selected for comparison of experimental data. There are very few previous papers about Python malicious code detection, so we will use webshell detection-related methods to reproduce the experiment. Moreover, due to the limited dataset, the deep neural network method may cause overfitting and cannot achieve better results, so the classic machine learning models are used.

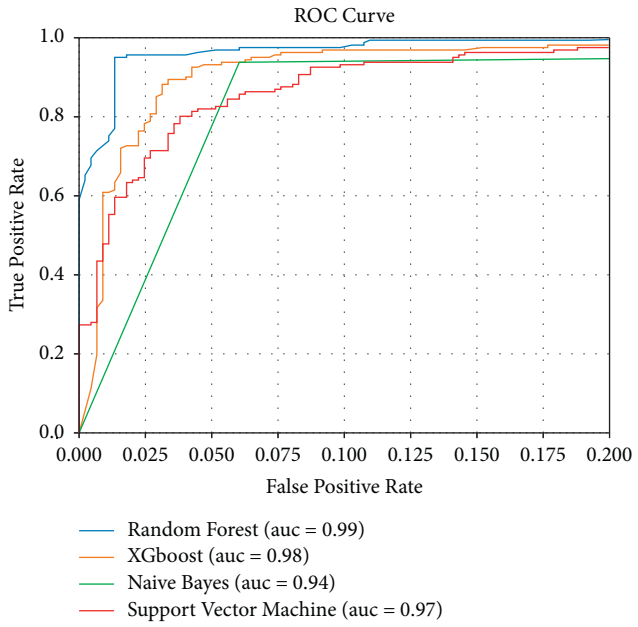

Figure 4: ROC curves of different algorithms.

Fang et al. [22] used FastText to detect webshell. The method is similar to the synthesis of feature 2 and FastText features in Section 4.3.2. The difference is that the statistical text feature of compression rate is not considered, and $n = 4$ is selected for n-gram value of FastText, while $n = 5$ is used in this paper. Cui et al. [18] used Random Forest and Gradient Boosting Decision Tree comprehensive algorithms to distinguish webshells. They also considered six types of text statistical features. Besides, they used TF-IDF and Hash, two types of vectors to represent opcode sequences, and obtained sequence classification labels through Random Forest classifiers. After synthesizing the first six types of features, the final prediction result was obtained through the GBDT classifier. Unlike this paper, TF-IDF represents the frequency of a single character. Guo et al. [38] recognized webshell attacks through opcode and also used TF-IDF to represent text. However, the bi-gram was used to divide characters, and the final classifier chose Naive Bayes (the method of the paper below is represented by the author's last name).

We use the dataset of this paper to reproduce the above three experiments, and compare the performance with PBDT. The results are shown in Table 7. It should be noted that most of the parameter selection in the experiment is described in the original text, but some adjustments are made due to different applicable scenarios. For the part related to the text's statistical characteristics, the first two papers are aimed at PHP webshells. The dangerous characters proposed are also related to the PHP language. We will replace them with the Python backdoor dangerous keywords listed in Section 3.2.6. The TF-IDF vector dimension chosen by Cui [18] is 146 because it is for a single character, and there are 146 types of opcodes in PHP. There are 121 types of Python opcodes in this experiment, so the vector dimension is set to 121.

Table 7: Performance comparison with other models.

	Accuracy	Precision	Recall (TPR)	TNR	F1-score
Guo et al. [38]	0.8355	0.6393	0.8696	0.8233	0.7368
Cui [18]	0.9122	0.8961	0.7565	0.9682	0.8201
Fang et al. [22]	0.9391	0.8563	0.9255	0.9441	0.8896
PBDT	0.9770	0.9623	0.9503	0.9866	0.9563

The table's data intuitively shows that PBDT is significantly better than the other three solutions. In-depth analysis, the algorithm used by Guo et al. [38] is Naive Bayes. In Section 4.3.3, we have compared the algorithms, and the Random Forest performs better in the classification of Python backdoors. Random Forest is a combination of multiple decision trees with strong generalization ability. At the same time, the risk of overfitting is reduced by averaging decision trees, and it performs well for the classification of high-dimensional data. The Naive Bayes model assumes that the attributes are independent of each other, and the classification effect is not good when the number of attributes is relatively large or the correlation between attributes is relatively large. The classification object of this paper is the entire Python file. File sizes and structures vary greatly. Different malicious files may have different manifestations, and the location of suspicious statements in the code is uncertain. At the same time, the feature dimension used is higher, and the internal correlation of similar features is also greater. Therefore, compared with the Naive Bayes used by Guo et al. [38], Random Forest is more suitable for the sample scenario in this paper. Simultaneously, the bi-gram only expresses the relationship between the adjacent opcodes, and the opcode to express a complete sentence of Python language needs five or more, so the $n = 5$ used in this paper can better represent the semantic information of the text. The reason for the significantly lower precision of this scheme is the imbalance in the number of positive and negative samples.

The same is true for the selection of n in Fang et al.'s [22] FastText model. We have proved through a lot of experiments that $n = 5$ can guarantee the best model performance. The TF-IDF vector and hash vector in Cui [18] only represent the mapping relationship between a single opcode and the vector and do not reflect the contextual connection. Therefore, the classification effect is not excellent. The essence of n-gram is to divide the sequence of opcodes into the smallest subblocks that can represent code information. It takes 5 or more to express a complete sentence of opcodes in the Python language. For example, for the commonly used socket connection statement "socket.socket (socket.AF_INET, socket.SOCK_STREAM)" in the backdoor, the corresponding opcode sequence is "["LOAD_NAME", "LOAD_METHOD", "LOAD_NAME",

"LOAD_ATTR", "LOAD_NAME", "LOAD_ATTR", "CALL_METHOD", "RETURN_VALUE"]". The number of opcodes is 8, and the more parameters in the function call, the longer the sequence length. Consequently, the value of n

should not be too small. Cui's [18] method is similar to the value $n = 1$, and Fang et al.'s [22] method is $n = 4$, neither of which can effectively represent the code semantic information. Therefore, the value of n in this paper is reasonable and effective for classification.

In summary, compared with previous research, PBDT can better identify malicious Python backdoor.

5. Conclusion

In order to ensure the concealment of backdoor, the attacker will obfuscate and encode the code. Concurrently, the parts of the text that are not related to the function will also affect the detection effect. But the encoding often has apparent characteristics, and the interference items such as comments will not be compiled. This paper proposes and constructs a Python backdoor detection model PBDT, representing the text through the statistical features caused by obfuscation and the features of opcode sequence in the compilation, and matches the suspicious modules and functions in the code as well. The above features can be used for backdoor recognition, respectively. However, experiments have proved that the detection effect of comprehensive features is the best. When using the Random Forest classifier, the accuracy of 97.70% and the TNR of 98.66% can be obtained.

Compared with the dynamic detection which requires high detection environment and the deep learning that consumes a lot of resources, the static detection scheme based on machine learning proposed in this paper can obtain better detection results with limited resources. What is more, under the premise that the dataset contains some obfuscated samples, it has a significant performance improvement compared with the previously proposed webshell's detection method. However, this scheme covers many aspects and has an extensive feature dimension. How to obtain better detection performance with limited feature dimensions is a direction worthy of our future research. At the same time, exploring the characteristics of other programming language's backdoor scripts is also a valuable work. The design idea of this paper is also applicable to other malicious Python code detection.

Data Availability

The data used to support the findings of this study are included within the article.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this study.

Acknowledgments

This paper was supported in part by the National Natural Science Foundation of China (U20B2045) and National Key Research and Development Program of China (Grant no. 2016QY13Z2302).

References

[1] Malwarebytes Labs, "2020 state of malware report," 2020, avaible at: https://resources.malwarebytes.com/files/2020/02/2020_State-of-Malwarebytes.Report.pdf.

[2] Y. D. Lawrence, D. Liang Lee, Y.-H. Chen, and L. Xiang Yann, Lexical analysis for the webshell attacks," in *Proceedings of the 2016 International Symposium On Computer, Consumer And Control (IS3C)*, pp. 579–582, IEEE, Xi'an, China, July 2016.

[3] O. Starov, J. Dahse, S. S. Ahmad, T. Holz, and N. Nikiforakis, "No honor among thieves: a large-scale analysis of malicious web shells," in *Proceedings of the 25th International Conference on World Wide Web*, pp. 1021–1032, Montréal, Canada, April 2016.

[4] C. Wang, H. Yang, Z. Zhao, L. Gong, and Z. Li, "The research and improvement in the detection of PHP variable webshell based on information entropy," *Journal of Computers*, vol. 28, pp. 62–68, 2016.

[5] M. Peter and B. V. W. Irwin, "Towards a PHP webshell taxonomy using deobfuscation-assisted similarity analysis," *IEEE*, in *Proceedings of the 2015 Information Security for South Africa (ISSA)*, pp. 1–8, Johannesburg, South Africa, August 2015.

[6] H. Zhang, M. Liu, Z. Yue, Z. Xue, Y. Shi, and X. He, "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)*, pp. 1584–1591, IEEE, Guangzhou, China, January 2020.

[7] Z. Zhang, M. Li, L. Zhu, and X. Li, "Smartdetect: a smart detection scheme for malicious web shell codes via ensemble learning," in *Proceedings of the International Conference on Smart Computing and Communication*, Tokyo, Japan, December 2018.

[8] Z. Zhao, Q. Liu, T. Song, Z. Wang, and X. Wu, "WSLD: detecting unknown webshell using fuzzy matching and deep learning," in *Proceedings of the International Conference on Information and Communications Security*, pp. 725–745, Springer, Beijing, China, December 2019.

[9] D. Canali and D. Balzarotti, "Behind the scenes of online attacks: an analysis of exploitation behaviors on the web," in *Proceedings of the 20th Annual Network & Distributed System Security Symposium (NDSS 2013)*, San Diego, CA, USA, February 2013.

[10] M. Xie and J. Hu, "Evaluating host-based anomaly detection systems: a preliminary analysis of adfa-ld," in *Proceedings of the 2013 6th International Congress on Image and Signal Processing (CISP)*, vol. 3, pp. 1711–1716, IEEE, Hangzhou, China, December 2013.

[11] Z.-H. Lv, H.-B. Yan, and R. Mei, "Automatic and accurate detection of webshell based on convolutional neural network," in *Proceedings of the China Cyber Security Annual Conference*, pp. 73–85, Springer, Beijing, China, August 2018.

[12] Y. Tian, J. Wang, Z. Zhou, and S. Zhou, "CNN-webshell: malicious web shell detection with convolutional neural network," in *Proceedings of the 2017 6th International Conference on Network, Communication and Computing*, pp. 75–79, Kunming, China, December 2017.

[13] Z. Wang, J. Yang, M. Dai, R. Xu, and X. Liang, "A method of detecting webshell based on multi-layer perception," *Academic Journal of Computing & Information Science*, vol. 2, p. 1, 2019.

[14] B. Yong, X. Liu, Y. Liu, H. Yin, L. Huang, and Q. Zhou, "Web behavior detection based on deep neural network," in

Proceedings of the 2018 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI), pp. 1911–1916, IEEE, Guangzhou, China, October 2018.

[15] S. L. Thomas and A. Francillon, "Backdoors: definition, deniability and detection," in *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions, and Defenses*, pp. 92–113, Springer, Heraklion, Greece, September 2018.

[16] B. Scott and B. Hagen, "Web shell detection using NeoPI," 2011, https://resources.infosecinstitute.com/topic/web-shell-detection/.

[17] A. Fass, M. Backes, and B. Stock, "JStap: a static pre-filter for malicious javascript detection," in *Proceedings of the 35th Annual Computer Security Applications Conference*, pp. 257–269, San Juan, PR, USA, December 2019.

[18] H. Cui, "Webshell detection based on random forest–gradient boosting decision tree algorithm," in *Proceedings of the IEEE 3rd International Conference on Data Science in Cyberspace (DSC)*, June 2018.

[19] J. Wang, Y. Xue, Y. Liu, and H. Tian, "Jsdc: a hybrid approach for javascript malware detection and classification," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pp. 109–120, Singapore, April 2015.

[20] T. D. Tu, G. Cheng, X. Guo, and W. Pan, "Webshell detection techniques in web applications," in *Proceedings of the 5th International Conference on Computing, Communications and Networking Technologies (ICCCNT)*, pp. 1–7, IEEE, Hefei, China, July 2014.

[21] I. A. Al-Taharwa, H.-M. Lee, A. Jeng, K. Wu, C. Ho, and S. Chen, "JSOD: javascript obfuscation detector," *Security and Communication Networks*, vol. 8, no. 6, pp. 1092–1107, 2015.

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

[23] Z. Pan, Y. Chen, Y. Chen, Y. Shen, and X. Guo, "Webshell detection based on executable data characteristics of PHP code," *Wireless communications and mobile computing*, vol. 2021, Article ID 5533963, 12 pages, 2021.

[24] Y. Wu, Y. Ma, and S. Wan, "Multi-scale relation reasoning for multi-modal visual question answering," *Signal Processing: Image Communication*, vol. 96, Article ID 116319, 2021.

[25] S. Ding, S. Qu, Y. Xi, and S. Wan, "Stimulus-driven and concept-driven analysis for image caption generation," *Neurocomputing*, vol. 398, pp. 520–530, 2020.

[26] S. Ndichu, S. Ozawa, T. Misu, and K. Okada, "A machine learning approach to malicious JavaScript detection using fixed length vector representation," in *Proceedings of the 2018 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, IEEE, Rio de Janeiro, Brazil, July 2018.

[27] H. C. Kim, Y. H. Choi and H. L. Dong, JsSandbox: a framework for analyzing the behavior of malicious JavaScript code using internal function hooking," *KSII Transactions on Internet & Information Systems*, vol. 6, p. 2, 2012.

[28] K. Rieck, T. Krueger, and A. Dewald, "Cujo: efficient detection and prevention of drive-by-download attacks," in *Proceedings of the 26th Annual Computer Security Applications Conference*, pp. 31–39, Austin, TX, USA, December 2010.

[29] A. Fass, M. Backes, and B. Stock, "Hidenoseek: camouflaging malicious javascript in benign asts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1899–1913, London, UK, November 2019.

[30] A. Fass, R. P. Krawczyk, M. Backes, and B. Stock, "JaSt: fully syntactic detection of malicious (obfuscated) javascript," in *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 303–325, Springer, Saclay, France, June 2018.

[31] A. Kapravelos, S. Yan, M. Cova, C. Kruegel, and G. Vigna, "Revolver: an automated approach to the detection of evasive web-based malware," in *Proceedings of the 22nd USENIX Security Symposium (USENIX Security*, vol. 13, pp. 637–652, Washington, DC, USA, August 2013.

[32] L. Yu, J. Huang, I. Ademola, M. Mitchell, J. Zhang, and R. Dai, "Shellbreaker: automatically detecting PHP-based malicious web shells," *Computers & Security*, vol. 87, Article ID 101595, 2019.

[33] Z. Li, A. Qi, C. Xiong, Y. Chen, T. Zhu, and H. Yang, "Effective and light-weight deobfuscation and semantic-aware attack detection for powershell scripts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1831–1847, London, UK, November 2019.

[34] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.

[35] J. Ramos, "Using tf-idf to determine word relevance in document queries," in *Proceedings of the 1st Instructional Conference On Machine Learning*, vol. 242, pp. 133–142, New Jersey, NJ, USA, August 2003.

[36] J. Armand, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of tricks for efficient text classification," 2016, https://arxiv.org/abs/1607.01759.

[37] T. K. Ho, "Random decision forests," in *Proceedings of the 3rd international conference on document analysis and recognition*, vol. 1, pp. 278–282, Montreal, Canada, August 1995.

[38] Y. Guo, H. Marco-Gisbert, and K. Paul, "Mitigating webshell attacks through machine learning techniques," *Future Internet*, vol. 12, p. 1, 2020.