TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 10/20 pp766–783
DOI: 10.26599/TST.2023.9010020
Volume 29, Number 3, June 2024

SmartEagleEye: A Cloud-Oriented Webshell Detection System Based on Dynamic Gray-Box and Deep Learning

Xin Liu, Yingli Zhang, Qingchen Yu, Jiajun Min, Jun Shen, Rui Zhou*, and Qingguo Zhou*

Abstract: Compared with traditional environments, the cloud environment exposes online services to additional vulnerabilities and threats of cyber attacks, and the cyber security of cloud platforms is becoming increasingly prominent. A piece of code, known as a Webshell, is usually uploaded to the target servers to achieve multiple attacks. Preventing Webshell attacks has become a hot spot in current research. Moreover, the traditional Webshell detectors are not built for the cloud, making it highly difficult to play a defensive role in the cloud environment. SmartEagleEye, a Webshell detection system based on deep learning that is successfully applied in various scenarios, is proposed in this paper. This system contains two important components: gray-box and neural network analyzers. The gray-box analyzer defines a series of rules and algorithms for extracting static and dynamic behaviors from the code to make the decision jointly. The neural network analyzer transforms suspicious code into Operation Code (OPCODE) sequences, turning the detection task into a classification problem. Comprehensive experiment results show that SmartEagleEye achieves an encouraging high detection rate and an acceptable false-positive rate, which indicate its capability to provide good protection for the cloud environment.

Key words: Webshell; detection; cloud; web security; deep learning

1 Introduction

Cloud computing technology, which is characterized by its manageability, scalability, and availability, provides efficient and convenient resources for end users. The most widely used definition of the cloud computing model is introduced by NIST as “a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction”[1]. The rapid development of cloud computing has effectively promoted the Internet, while also inducing several new network security problems[2, 3].

Attack methods are endless but often have the same goal, that is, to steal and destroy the target’s data. One of the foundations for completing these attacks is to insert backdoor code in the cloud web system to establish persistent access to achieve the purpose of the episode. This backdoor code, Webshell, is a web-based implementation of the shell concept and can be uploaded on servers, either on clouds or edges, by malicious attackers[4]. Attackers could then escalate

• Xin Liu, Yingli Zhang, Jiajun Min, Rui Zhou, and Qingguo Zhou are with School of Information Science and Engineering, Lanzhou University, Lanzhou 730000, China. E-mail: xliu2019@lzu.edu.cn; zhangyingli2020@lzu.edu.cn; minjj18@lzu.edu.cn; zr@lzu.edu.cn; zhouqg@lzu.edu.cn.
• Qingchen Yu is with College of Computer Science and Technology, Zhejiang University, Hangzhou 310058, China. E-mail: qingchenyu@zju.edu.cn.
• Jun Shen is with School of Computing and Information Technology, University of Wollongong, Wollongong 2500, Australia. E-mail: jshen@uow.edu.au.
∗ To whom correspondence should be addressed.

Manuscript received: 2023-01-16; revised: 2023-03-18; accepted: 2023-03-19

privilege and maintain persistent remote access to victim servers through Webshells for future exploitation, such as command execution, database connection, network surveillance, and exposure of compromised servers to danger. A report[5] has indicated that exploitation of breaches within web applications remains a severe threat. The survey in this report revealed that 39% of the sites could be illegally accessed.

Attackers can mix the Webshell into the highly flexible normal code. They can also use encryption, encoding, and other methods to reduce the readability of Webshells, which substantially increases the difficulty of detection. Highly frequent data interactions and additional devices in the cloud environment provide increasing opportunities for attackers to implant Webshells. The widespread adoption of cloud technologies also amplifies the impact of security incidents and allows security issues to permeate across business processes.

Therefore, efficient solutions are demanded to handle such security breaches. A Webshell detection system based on PHP as a prototype is proposed to secure the cloud environment to solve the aforementioned problems. The core implementation components include a gray-box analyzer, a neural network analyzer, and a cloud platform. The white-boxed part in the gray-box analyzer is responsible for conducting taint analysis to check the possible flow of remotely-controlled taints into sensitive "sinks" and performing a preliminary inspection for code obfuscation. More importantly, this part extracts behaviors from the source code, of which the detailed definition and implementation can be found in Section 3.3. The black-boxed part in the gray-box analyzer is implemented as a PHP extension that extracts dynamic behaviors by hooking into the PHP kernel. In both pieces, code branches created by control statements, such as "if", are combined into the main branch to enhance code coverage.

The neural network analyzer is designed by the following the idea of code classification. The PHP source files are first transformed into OPCODE sequences[6]. A network based on Bidirectional Encoder Representation from Transformers (BERT)[7] and Bi-directional Long Short-Term Memory (BiLSTM), named Conv-BiLSTM, is designed to classify the OPCODE sequences.

A cloud-based distributed architecture ensures the stable and effective operation of the entire system in the cloud environment. This architecture can provide secure, convenient, fast, scalable, and continuously evolving Webshell defense capabilities for cloud environments with additional computing requirements and devices. This architecture is introduced later in Section 3.

Overall, the contributions of this work are as follows:

- A "gray-boxed" method of PHP-based Webshell detection, which mainly extracts behaviors from static source code and dynamic execution traces, is introduced. These behaviors are deliberately defined to be security-related to capture the maliciousness of Webshells.
- A learning-based neural network analyzer named Conv-BiLSTM, which uses OPCODE sequences of PHP source code that can reflect whether it is malicious or not, is designed.
- SmartEagleEye, a cloud-based, client-server distributed architecture with the gray-box and neural network analyzers as its core detection modules, is designed and implemented to protect the security of the cloud environment.
- A series of evaluation experiments for SmartEagleEye on Webshell detection is performed. Compared with other recognized server-based security softwares, the proposed system achieves superior detection accuracy and acceptable system overhead in the meantime.

The background of this paper, including basic knowledge of Webshells and intuition behind the system design, is described in Section 2. The design and implementation details of the SmartEagleEye system are introduced in Sectionn 3, outlining the structure of SmartEagleEye with four basic components: the platform server, the agent, the gray-box analyzer, and the neural network analyzer. Dataset collection and evaluation processes are described in Section 4. The paper is concluded and the future work is discussed in Section 5. Related work is introduced in Section 6.

2 Background

2.1 Webshell

Webshells usually work after "installation", so to speak, being uploaded onto the target server and maintaining persistent existence. Webshell is a type of web application written in supported languages of a target web server, which can be manipulated by attackers to enable remote access and administration of the compromised host. Numerous approaches can be used to upload Webshells, such as file inclusion, arbitrary file upload, and SQL injection. The consequences include keeping illegal remote accession, stealing credential information, hijacking the host as bots in botnets, or defacing the website with phishing intent or simply for a hoax. Figure 1 shows the threat model of the Webshell.

2.2 Code obfuscation

Developers usually use code obfuscation techniques to protect legitimate codes from reverse engineering and modification. Covering the true malicious intention naturally becomes impossible for attackers. The most popular paradigm is based on the encoder decoder. In that way, a malicious code payload is encoded (or decoded) into a string wrapped with the corresponding decode (in that way, a malicious code payload is encoded or decoded into a string that is wrapped with the corresponding decode or encode function) function.

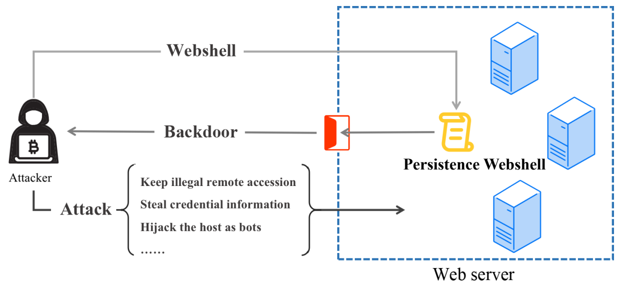


Fig. 1 Threat model.

Encoded payload with a decoder wrapper is usually concatenated upon some functions or language constructs, such as eval, require, assert, or preg_replace with “e” operation (deprecated since PHPv5.5.0). Indexes 1 and 2 in Table 1 are typical examples. The string is derived from executable payloads, and some encoding methods, such as Base64, intrinsically expand the size (3 bytes in and 4 bytes out for Base64). Therefore, the string is often eccentrically large.

Webshells can have various deformation techniques due to the flexible syntax and numerous built-in functions of PHP. Table 1 shows other viable methods. With loose syntactic constraints, slippery attackers may elaborate variable functions, which are substantially applying the same mechanism as the aforementioned ones, except that the real effective parts are stored in string variables and further concatenated together. Code snippet illustrates a direct use case by Index 3. Index 4 is also an exciting example. The final composition assert ($_POST[_]) can be obtained by performing XOR operations on non-alphabetic and non-numeric characters. Taking values from declared strings to spell out sensitive functions is also a way, as shown by Index 5. Code obfuscation has remained to be an intractable problem due to its tricky and various approaches.

2.3 PHP OPCODE

The use of PHP as a prototype is investigated in this section. Theoretically, Webshells could be implemented in any server-side programming language (e.g., PHP, ASP, JAVA, and Python); however, PHP is

Table 1 Examples of obfuscation.

Index	Code example
1	eval (base64_decode(“ZWNobyAiaGkiOw==”))
2	eval (gzuncompress (base64_decode (“eJxLTc7IV1D3yFQHAA8tAr8=”)));
3	$func1=‘as’.‘se’.‘rt’; $func2=‘base’.‘64’.‘_de’.‘code’; $code=‘cHJpbnQgImhpIjs=’; $func1 ($func2 ($code));
4	$_= (‘%01’^‘`’).(‘%13’^‘`’).(‘%13’^‘`’).(‘%05’^‘`’).(‘%12’^‘`’).(‘%14’^‘`’); $_ _=‘_’.(‘%0D’^‘]’).(‘%2F’^‘`’).(‘%0E’^‘]’).(‘%09’^‘]’); $_ _ _=$$_ _; $_($_ _ _[_]);

5	$sF=“PCT4BA6ODSE_”;
	$s21=strtolower ($sF[4].$sF[5].$sF[9].$sF[10].$sF[6].$sF[3].$sF[11].$sF[8].$sF[10].$sF[1].$sF[7].$sF[8].$sF[10]);
	$s22=$ {strtoupper ($sF[11].$sF[0].$sF[7].$sF[9].$sF[2])}[‘n985de9’];
	if (isset($s22)){
	    eval ($s21 ($s22));
	}

the most commonly used one partially due to its ubiquitous existence today[8]. According to the statistics by W^{3}Techs, PHP is used by nearly 80% of all the websites with known server-side programming languages, demonstrating that PHP has been one of the most widely used script languages on the Web[9]. PHP offers ease-of-use features for developers and malicious attackers because of its dynamic syntax and several built-in features. Therefore, this paper mainly focuses on Webshells developed in PHP languages.

A PHP source file can be compiled into a series of operation codes, commonly known as OPCODE[6]. An OPCODE is a numeric identifier of a single operation performed by the Zend Virtual Machine (Zend VM). This code can also be dumped by Vulcan Logic Dumper (VLD) extension, which hooks into the Zend engine. Table 2 shows the corresponding OPCODE, and integer sequences of a typical “one-sentence”' Webshell are shown in Fig. 2.

3 Design of SmartEagleEye

The stealthy nature of Webshells makes it hard to be distinguished from normal codes, introducing potential risks. A Webshell detection system named SmartEagleEye is proposed to address this problem. This section details the implementation of this system, from a high-level architecture overview to explaining each submodule.

SmartEagleEye is a distributed multiserver system comprising a platform server, database server, message queue server, gray-box analyzer, neural network analyzer, and agent. The structure overview is shown in Fig. 3. The following components work together to form a SmartEagleEye instance.

Table 2 OPCODE sequence of the example code.

Index	OPCODE
1	BEGIN_SILENCE
2	FETCH_R
3	FETCH_DIM_R
4	END_SILENCE
5	INCLUDE_OR_EVAL
6	RETURN

```
<?php
eval (@$POST['a']);
?>
```

Fig. 2 Example of a typical Webshell.

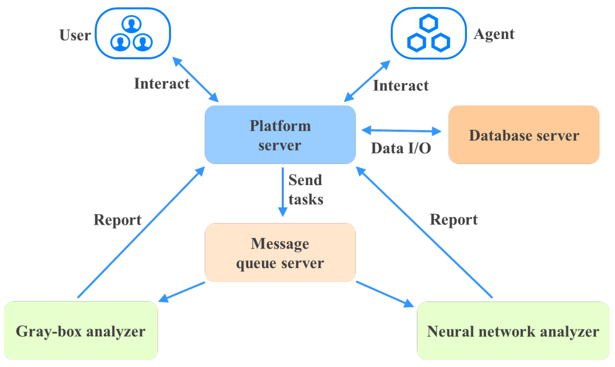


Fig. 3 Architecture overview of SmartEagleEye.

- The platform server is an essential interaction center in the SmartEagleEye system. This server provides RESTful API and a front-end interaction interface for administrators.
- The database server stores user, alarm, and processing data of the entire SmartEagleEye system.
- The message queue server distributes Webshell detection tasks to the gray-box and neural network analyzers for Webshell detection.
- The gray-box analyzer uses the dynamic gray-box method for Webshell detection of suspicious file samples.
- The neural network analyzer uses the Conv-BiLSTM model for Webshell detection of suspicious file samples.
- Agents are deployed on servers that must be protected and are used for initial detection, reporting, isolation of malicious samples, and monitoring server operating status.

The final detection result is jointly determined by the gray-box analyzer and the neural network detector. Specifically, the union of the two detector results is taken as the final detection result.

3.1 Platform server

The platform server is the operating hub of the SmartEagleEye system. The HTTPS protocol has a stateless nature; therefore, the platform server must accept the polling of the agent and afford heavy I/O pressure. Distributed technology provides centerless network based computer processing, which is the opposite of centralized technology. SmartEagleEye is a centralized client-server-based system, but the platform server is distributed on the basis of load balancing technology. Figure 4 shows the architectural design of the platform server, which is an independent device. However, this device comprises several web servers, a

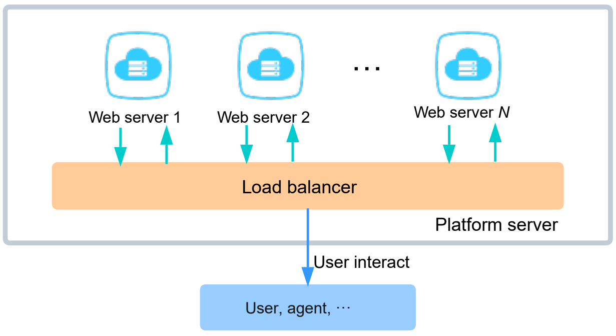


Fig. 4 Design of platform server.

load balancing server, and a health check server that provides complementary services.

Network I/O is intensive on the platform server. Therefore, SmartEagleEye uses the weighted least-connection scheduling algorithm for load balancing. The algorithm is described as follows.

First, the following is observed for cluster S with n servers.

$$S = \{S_1, S_2, \ldots, S_n\}, n \in \mathbf{N} \tag{1}$$

Suppose that the weight for server S_i in cluster S, is $W(S_i)$, and its number of connections is $C(S_i)$. The total number of connections in cluster S is then computed as follows:

$$C_{\text{sum}} = \sum_{i=1}^{n} C(S_i) \tag{2}$$

For server $S_t \in S$, if S_t satisfies

$$(S_t/C_{\text{sum}})/W(S_t) = \min\{(C_i/C_{\text{sum}})/W(S_i)\} \tag{3}$$

then C_{sum} is determined. Thus, simplifying the formula yields

$$S_t/W(S_t) = \min\{C_i/W(S_i)\} \tag{4}$$

The server S_t with the least weighted connection identified by the above algorithm would be the actual hosting server of the next connection. In addition, all web servers should check their health status and load balancing to ensure the high availability of platform servers. The faulty machine will be taken offline once any fault occurs.

3.2 Agent

The agent is the actual executor of SmartEagleEye to protect the servers that deploy the edge computing systems. In SmartEagleEye, the agent is responsible for monitoring web directory changes, reporting suspicious script files, collecting and reporting server operating status, and performing server security baseline checks.

Figure 5 shows the typical detection process of an agent. First, the agent obtains file change information through Linux inotify, then it performs RegEx-based detection on the target through a RegEx library stored locally once it senses a file change. If the agent cannot mark the sample as a Webshell by local detection, then the sample will be added to the queue. When the system is idle, the reporting queue is uploaded to the platform server for detection based on the gray-box and neural network analyzers. If the SmartEagleEye system finally determines that the sample is malicious, then the agent will either raise the alarm or immediately delete it according to the administrator scheme.

The agent then collects running data on the host and reports it to the platform server to assist administrators in monitoring the cloud server status. These data include basic operation and maintenance information (such as CPU usage, memory usage, and real-time network speed), as well as security baseline information (such as SELinux, web directory permission configuration, and web server permission configuration). Pretty Good Privacy (PGP) is an agreement used to encrypt information transmission and verify the origin and integrity of the data. This agreement comprises algorithms, such as hash, symmetric key, and asymmetric key. SmartEagleEye uses PGP for agent update signing and verification.

3.3 Grey-box analyzer

"Grey-box" means that this module takes advantage of black-boxed dynamic and white-boxed static code analysis techniques to make the decision jointly. The

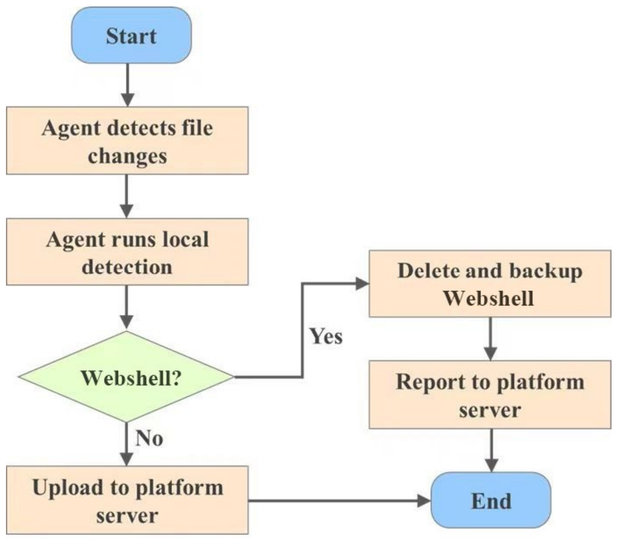

Start
Agent detects file changes
Agent runs local detection
Webshell?
Yes
No
Delete and backup Webshell
Report to platform server
Upload to platform server
End

Fig. 5 Detection process of agent.

core idea is to extract behavior from the code to distinguish malicious from normal. This study mainly focuses on security issues. Therefore, increased attention is provided to behaviors that reflect malicious intentions.

The following assumptions regarding the production environment are presented in accordance with practical experiences. This module is designed on the basis of these assumptions.

- Web applications should not be heavily modified by their own or other programs in a production environment where the deployment has been completed.
- If the execution trace of a program shows inconsistency with its appearance, then this program is likely attempting to hide its true intention.

Based on the two assumptions, the agent of SmartEagleEye is set to start monitoring the file system only after the user confirms that all the environment deployment in the operating system has been completed. This mode effectively reduces the number of files that must be analyzed. This strategy avoids false positives caused by third-party encrypted libraries, and empowers the gray-box analyzers to use aggressive analysis strategies for edge computing systems.

The gray-box analyzer has no special requirement for expensive hardware such as GPUs. As shown in Fig. 6, every gray-box analyzer instance is encapsulated in a lightweight virtualized environment that currently uses docker containers. Thus, multiple gray-box analyzers can work simultaneously on a single server. This distributed architecture effectively enhances the processing capabilities of the SmartEagleEye system while ensuring its security.

As shown in Fig. 7, the architecture of the gray-box analyzer mainly includes four parts: message queue module, static analysis engine, dynamic analysis engine, and behavior analysis engine. The role of the message queue module is to obtain file locations of samples under detection, download sample files, and report detection results through the message queue server. The module is a Python-based AMQP protocol[10], where the consumer client communicates directly with the behavior analysis engine and message queue server.

The static analysis engine builds an Abstract Syntax Tree (AST) from the source code and analyzes the taint spread and function calls of the sample based on the AST. The dynamic analysis engine is responsible for adding hooks to the PHP kernel through PHP extensions and uses these hooks to trace behaviors such as function calls in samples. The behavior analysis module is designed to analyze the behaviors of PHP samples according to predetermined rules by synthesizing and tracing results of the static and dynamic behavior engines, and then determine whether the sample is a malicious Webshell.

3.3.1 Sample preprocessing

Directly executing a program cannot guarantee the tracing of every branch of the code due to the control statements (e.g., “if-else” and “switch”). Therefore, the gray-box analyzer merges branches generated by

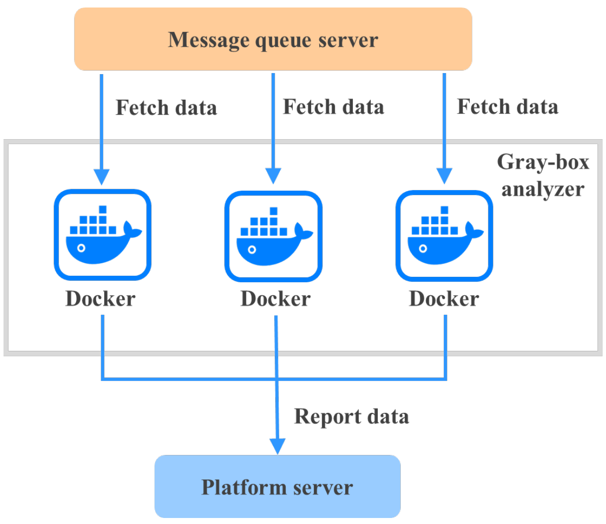


Fig. 6 Implementation of container.

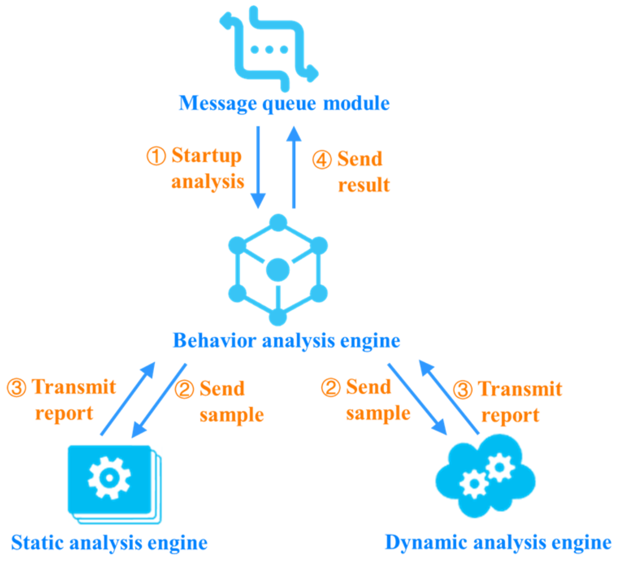


Fig. 7 Architecture of gray-box analyzer.

multiple control statements into one branch, improving dynamic testing coverage. The merged sample has different data and control flow structures compared with the original ones, but the order and number of behaviors are changed. By contrast, behaviors, such as assignment, reading, function definition, function call, and PHP code constructor call, are not modified. In the implementation, the behaviors of the branch-merged samples are used to approximate the original ones and perform further analysis.

Samples are also formatted with the PSR-12 standard[11] in sample preprocessing.

3.3.2 Static analysis engine

First, the source code is converted into an AST. Then, the static analysis engine traverses the AST, tracing and recording suspicious behaviors. The results of the recorded behavior are fed into the behavior analysis engine after the traversal is completed. A vast majority of Webshells are built with process-oriented programming. Thus, the current version of the static analysis engine in SmartEagleEye does not have full object-oriented analysis support. However, the gray-box analyzer also has a good detection effect for most Webshells developed on the basis of object-oriented programming due to the aggressive detection strategy.

Static behavior analysis starts after AST extraction. The gray-box analysis object is the behavior of the PHP scripts; therefore, the behavior must be recorded in a certain data structure. First, the set S^{static} is defined as the specific suspicious behavior set of the PHP script. S_i^{static} is the i-th suspicious behavior traced by the sample from the entry point of the program,

$$S_i^{\text{static}} = f(t_{\text{type}}, k, l, s, e) \tag{5}$$

The following five parameters can jointly determine a specific suspicious behavior. Therefore, a suspicious behavior data structure of the static analysis engine is defined as a five-tuple $(t_{\text{type}}, k, l, s, e)$.

- Variable t_{type} is the type of behavior. The types that our static analysis engines focus on include variable assignments, ordinary function calls, function definitions, and PHP code constructor calls such as eval ( ) and include ( ).
- Variable k is the name of the called function or PHP's code constructor. For assignment statements, k is set to null.
- Variable l is a list of behavior-related values. For function calls, it is the parameters list to be passed. For the assignment, it is a constant or variable that participates in the expression on the assignment operation's right side.
- Variable s is the line number where the behavior starts, and variable e is the line number where the behavior ends.
- The mapping f is an injection of the four aforementioned variables into S_i^{static}.

The traversal algorithm is shown in Algorithm 1. The input AST root represents the entry point to the program, behavior_set represents the extracted static behavior, and the later_list holds nodes that are then traversed separately for later analysis. The NextNode function is used to obtain the next node to be processed in the AST, the NeedTraverseLater function determines the order of traversals, and the ExtractBehavior function extracts the static behavior of the node. Overall, the static analysis engine focuses on the type of each node, the row number of the node, and related variables. The static analysis engine traversal logic is entered from the program entry point and traversed according to each statement. If the statement contains child nodes, it is determined according to the node type whether to start traversing its child nodes (such as code block statement nodes) or to iteratively analyze its children later (such as function definition nodes).

S^{static} is obtained after traversal, and the static analysis engine transfers S^{static} to the behavior analysis module through the Linux pipeline.

3.3.3 Dynamic analysis engine

Similar to the static analysis engine, a suspicious

Algorithm 1 Node traversal algorithm

```
Input: AST root
Output: Behavior set of the input AST
behavior_set ← ∅;
later_list ← ∅;
node ← AST root;
while NextNode (node) do
    if NeedTraverseLater (node) then
        later_list ← later_list ∪ node;
    else
        behavior_set ← behavior_set ∪ ExtractBehavior (node);
    end
end
for waiting_node ∈ later_list do
    behavior_set ← behavior_set ∪ ExtractBehavior
    (waiting_node);
end
Return behavior_set
```

behavior set for dynamic analysis is also defined in this study. The set S^{dynamic} is defined, and S_i^{dynamic} is the i-th suspicious behavior from the entry point of the program. Therefore,

$$S_i^{\text{dynamic}} = g(t_{\text{type}}, k, n_l) \tag{6}$$

The following three parameters can jointly determine a specific suspicious behavior. Therefore, a suspicious behavior data structure of the dynamic analysis engine is defined as a three-tuple (t_{type}, k, n).

- Variable t_{type} is the type of behavior. The types that the dynamic analysis engine focuses on are ordinary function calls and PHP code constructor calls, such as eval ( ) and include ( ).
- Variable k is the function name of ordinary function calls and PHP code constructor calls.
- Variable n_l is the line number where the behavior is detected. The PHP kernel provides this information.
- The mapping g is an injection of the above three variables into S_i^{dynamic}.

The goal of the dynamic analysis engine is to extract behavior while sampling the execution. The engine can effectively extract behaviors that are difficult to capture using static methods, e.g., whether the code has been obfuscated or not.

Dynamic analysis engine performs dynamic behavior tracing by monitoring and recording the actual function calls. The dynamic analysis engine only monitors and records function calls in PHP closely related to, or commonly used in, Webshell development due to performance reasons. Table 3 shows functions currently monitored and recorded by SmartEagleEye, and these functions are considered dangerous.

Dynamic analysis engine uses PHP extension with kernel hooks to monitor and record dynamic behaviors. Figure 8 shows two kinds of functions that must be traced.

- For built-in functions, the dynamic analysis engine first deletes the function from the PHP function table through the PHP extension, and then registers a new function with the same name. In this function, with the same name, the dynamic analysis engine records the function call according to the record. After recording, it returns the passed data to the original function for processing and continues tracking.
- For PHP code constructors, the dynamic analysis engine needs to override the zend_complie_string, which the Zend engine uses to convert strings to PHP code for execution.

Like the static analysis engine, the dynamic analysis engine transfers the monitered behaviors to a behavior analysis module through the Linux pipeline.

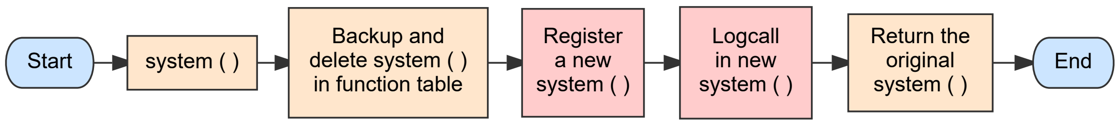
3.3.4 Behavior analysis engine

In addition to sample preprocessing, the behavior analysis engine must also synthesize results from static and dynamic analysis. These behavior records enable the behavior analysis engine to extract information further to analyze the sample files effectively. First, some sets of features are defined to assist analysis. These sets are unordered, including:

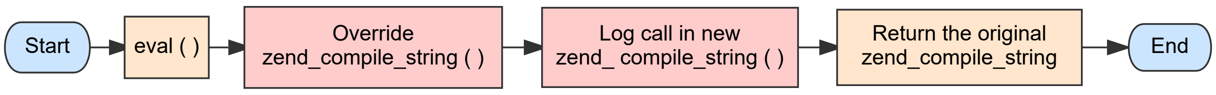
- $S_{\text{static\_construct\_line}}$: Set of PHP code constructors' line numbers accumulated by the static analysis engine.

Table 3 List of dangerous functions.

Name	Type	Name	Type
eval	Constructor	include	Constructor
require	Constructor	preg_replace	Built-in
unserialize	Built-in	extract	Built-in
system	Built-in	exec	Built-in
shell_exec	Built-in	passthru	Built-in
create_function	Built-in	call_user_func	Built-in
unserialize	Built-in	proc_open	Built-in
popen	Built-in	array_walk	Built-in
array_udiff	Built-in	preg_filter	Built-in
array_map	Built-in	array_walk_recursive	Built-in
uksort	Built-in	register_tick_function	Built-in
array_diff_ukey	Built-in	yaml_parse	Built-in
yaml_parse_url	Built-in	assert	Built-in
yaml_emit	Built-in	register_shutdown_function	Built-in
mb_ereg_replace	Built-in	array_diff_ukey	Built-in


Start
system ( )
Backup and delete system ( ) in function table
Register a new system ( )
Logcall in new system ( )
Return the original system ( )
End

(a) Example: system ( ) for dangerous function call


Start
eval ( )
Override zend_compile_string ( )
Log call in new zend_ compile_string ( )
Return the original zend_compile_string
End

(b) Example: eval ( ) for code constructor call

Fig. 8 Algorithm for tracing dynamic behavior.

- $S_{\text{static\_func\_line}}$: Set of function calls' line numbers accumulated by the static analysis engine.
- $S_{\text{static\_dangerous\_call}}$: Set of dangerous function calls accumulated by the static analysis engine. The dangerous functions are listed in Table 3 and $S_{\text{static\_dangerous\_call}} \subseteq S_{\text{static\_func\_line}}$.
- $S_{\text{static\_dangerous\_fvar}}$: Set of variables related to dangerous function calls and code constructor calls.
- $S_{\text{dynamic\_construct\_line}}$: the line number of constructor calls captured by the dynamic analysis engine.
- $S_{\text{dynamic\_func\_call}}$: Set of dangerous function calls accumulated by the dynamic analysis engine.

Attackers often change the control flows of Webshells to hide their true intention. However, collected features reflect the actual behavior, which has minimal effects on control flows. Therefore, detection evading methods can hardly succeed. These features are then used as input for rule-based Webshell detection.

Webshells are recognized with the blacklist policy after feature extraction. Therefore, the gray-box analyzer marks the sample as a Webshell once the behavior record of a sample meets one or more rules. The blacklist policy comes from a variety of practical security strategies, including:

(1) Constructor calls in dynamic analysis are different from ones in static analysis. For a sample that has finished branch merging and code formatting, the sample may be trying to hide its code construction behavior if its constructor calls are different from dynamic analysis and static analysis,

$$S_{\text{dynamic\_construct\_line}} \neq S_{\text{static\_construct\_line}} \tag{7}$$

(2) Dangerous function calls are found in dynamic analysis but not in static analysis. The design principle of this rule is similar to the previous one, both of which discover hidden calling behaviors, except that the rule's target is dangerous function calls,

$$\exists\, t \in S_{\text{dynamic\_func\_call}} \wedge t \notin S_{\text{static\_dangerous\_call}} \tag{8}$$

(3) Dangerous function calls contain variables that can be controlled by external code. First, the set S_{external} is defined as the set of externally controllable variables. This set contains all the externally controllable variables, including the GET, POST, and HTTP header sets transmitted by HTTP. If an external user controls a part of a dangerous function call (for example, part of POST data is used in the parameter of the system( ), then the call is likely to constitute a remote code execution vulnerability. Therefore, this sample is dangerous and highly suspected of Webshell. A similar phenomenon is if the code constructor calls contain external controllable variables,

$$\exists\, t \in S_{\text{external}} \wedge t \in S_{\text{static\_dangerous\_fvar}} \tag{9}$$

(4) Code constructor calls are related to a variable of abnormally large size. Generally, normal code does not have a particularly large variable, especially in code constructors. If a particularly large variable is found, the behavior analysis engine first checks whether the semantic separator frequency meets the natural language standard. If not, the behavior analysis engine will mark this variable as encrypted or obfuscated and considers the file a Webshell according to the second design principle of SmartEagleEye. This rule deals with Webshells that partially use asymmetric encryption algorithms to encrypt key code and dynamically execute actual code through PHP code constructors.

3.4 Neural network analyzer

The neural network analyzer is one of the core technologies of SmartEagleEye. An optimized network Conv-BiLSTM comprising three modules is proposed: the OPCODE extraction module, the word vector conversion module, and the detection module. The OPCODE extraction module converts the PHP file into OPCODE. The word vector module then extracts the text features for the first time and converts the text sequence into a word vector representation. In the detection module, multiple convolutional and pooling layers of different sizes are first utilized to perform further feature extraction on the text sequence. Afterward, the improved Bi-LSTM network is used to extract the features of the word vector sequence comprehensively through a two-way structure. The fully connected layer aggregates the text features according to different weights for normalization and classification through training on the pre-labeled text. Finally, the detection results of the Webshell and the regular PHP script files are obtained. An overview of the architecture is depicted in Fig. 9. A highly detailed part of the transformer structure in the word vector extraction layer can be found in Fig. 10.

3.4.1 OPCODE extraction

Codes must be represented with vectors before being fed into models. Therefore, code representation must be investigated. Naturally, it can be flattened into token sequences and then processed with sequence models,

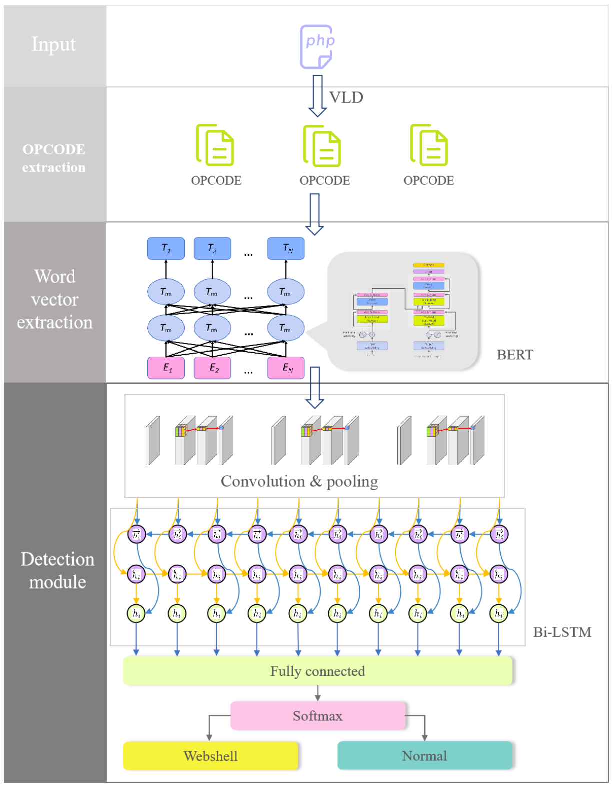


Fig. 9 Conv-BiLSTM architecture. In the word vector extraction layer, the variable T_i (i = 1, 2, …, N) represents the input variable of BERT, $T_{\rm rm}$ denotes the multilayer transformer encoder, and E_i (i = 1, 2, …, N) represents the output of BERT.

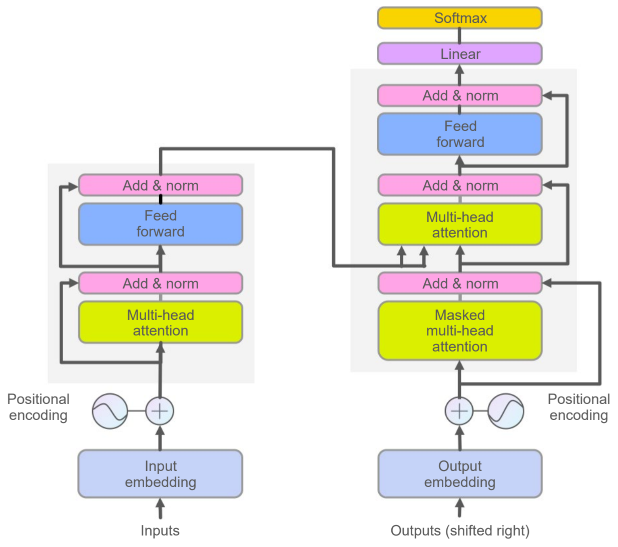


Fig. 10 Transformer architecture.

turning the malicious code detection into text classification[12]. A similar approach is adopted in the proposed system, which utilizes OPCODE instead of text tokens. Vulcan logic dumper is used on the basis of the Zend engine to implement the OPCODE output of PHP scripts employing kernel hooks.

3.4.2 Word vector conversion

BERT is then used to extract information from the OPCODE sequence and convert it into a word vector matrix. The final output vector contains the syntactic structure and semantic information between contexts to extract preliminary text information. The BERT language model synthesizes the relationship among multiple sentences in the text and the relationship among numerous words under a single sentence, and uses a multilayer transformer network[13] to obtain semantic information in the context of text sentences. An overview of the architecture is depicted in Fig. 10. The BERT language model completes the pretraining process through the Masked Language Model (MLM) and Next Sentence Prediction (NSP) combined with a substantial unlabeled corpus. The MLM model occludes 15% of randomly selected words in advance, and then uses the occlusion context to predict the comments. NSP then enables the model to capture the dependencies between sentences.

3.4.3 Detection

As the last module of the Conv-BiLSTM network, the detection module mainly completes the further extraction and classification tasks of OPCODE text sequence features. Through the two aforementioned sections, the initially extracted vectorized representation of the words containing the text features of the OPCODE sequence is used as the module input. The detection module employs an optimized Bi-LSTM that combines a multilayer convolution and pooling structure to differentiate between the Webshell and regular PHP script files to obtain the classification result. The structure of the detection module, including the word embedding layer, convolution layer, pooling layer, Bi-LSTM hidden layer, and fully connected layer, is shown in Fig. 9.

Inspired by the image feature extraction principle of Convolutional Neural Network (CNN), the convolutional and pooling layers in the network are used to feature OPCODE sequences further. Multiple convolution kernels and the processed word vectors are utilized in matrix multiplication operations at the corresponding positions, and then added to obtain a new word vector matrix. The linear model cannot effectively express the text characteristics during the convolution pooling operation. Thus, the activation function performs nonlinear processing on the outcome after the pooling operation. Simultaneously, this function prevents the convolution layer from producing completely different results due to the small changes in the word embedding layer. Afterward, instead of comprehensively retaining all the text information, the maximum pooling is selected to perform the pooling operation to extract the feature information in the text.

Some prominent Trojans could generate a long series. Therefore, LSTM networks[14], which can learn long-term dependencies, are selected as part of the detection model. The LSTM model comprises input words at time t, indicated by x_t, cell state C_t, temporary cell state, hidden layer state h_t, forget gate, memory gate, and output gate. By ignoring the information in the cell state and memorizing the new report, the useless data are discarded, the valuable information for subsequent calculations can be transmitted, and the state of the hidden layer h_t is outputted at every time step.

The first step in LSTM is to determine what information should be thrown away from the cell state. The decision is made by the sigmoid layer f_t called the "forget gate", which takes the hidden state value of the previous layer h_{t-1} and the input word at the current moment x_t as inputs. This layer outputs a number between 0 and 1 for each number in the cell state C_{t-1}. And the variables W and b represent the model's learnable parameters. In LSTM, function σ refers to the Sigmoid function, which performs gate control operations to regulate information flow and capture long-term dependencies,

$$f_t = \sigma\left(W_f\left[h_{t-1}, x_t\right] + b_f\right) \tag{10}$$

The next step is to decide what new information to be stored in the cell state. First, a sigmoid layer i_t, called the "input gate layer", determines which values will be updated. Next, a tanh layer creates a vector of new candidate values $\widetilde{C_t}$ that could be added to the state,

$$i_t = \sigma\left(W_i\left[h_{t-1}, x_t\right] + b_i\right) \tag{11}$$

$$\widetilde{C_t} = \tanh\left(W_C\left[h_{t-1}, x_t\right] + b_C\right) \tag{12}$$

Then, the determined things to be forgotten earlier are ignored by multiplying the old cell state C_{t-1} by f_t and adding $i_t \times \widetilde{C}$ to update the old cell state,

$$C_t = f_t \times C_{t-1} + i_t \times \widetilde{C_t} \tag{13}$$

Finally, the output is determined. A sigmoid layer o_t, which determines what parts of the cell state to be outputted, is run. Simultaneously, the cell state is put through a tanh layer and then multiplied by the outcome of the sigmoid layer,

$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right) \tag{14}$$

$$h_t = o_t \times \tanh\left(C_t\right) \tag{15}$$

The above introduction revealed that LSTM can effectively obtain the context information of the text. However, the Bi-LSTM with a bidirectional structure can process the historical knowledge of the text sequence and use the backward LSTM structure to capture the future information of the text sequence. Combining historical and future information corresponding to the two directions makes it possible to extract text sequence information more comprehensively, which plays a vital role in the classification task. The existing research content[15] shows that stacking multiple Bi-LSTM layers can enhance the predictive capability of the module. The overview of the architecture is depicted in Fig. 11. f-LSTM represents the forward LSTM unit, while b-LSTM stands for the backward LSTM unit. The formula for calculating the states of the f-LSTM unit and the b-LSTM are respectively presented as follows:

$$h_t^f = f\left(W^f\left[h_{t-1}^f, x_t\right] + b^f\right) \tag{16}$$

$$h_t^b = f\left(W^b\left[h_{t-1}^b, x_t\right] + b^b\right) \tag{17}$$

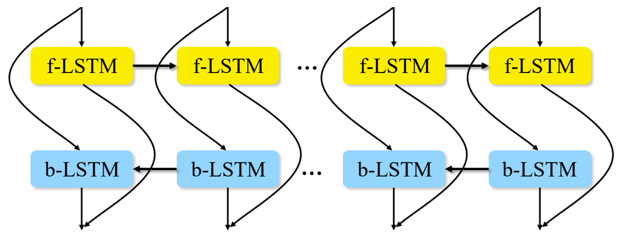


Fig. 11 BiLSTM architecture.

4 Experiment

This section evaluates SmartEagleEye through the following questions.

RQ1: Is it effective to use OPCODE as a Webshell detection feature?

RQ2: Can the neural network detector of SmartEagleEye effectively detect Webshells? Is it better than existing neural network models?

RQ3: Compared with industrial detection tools, does SmartEagleEye have superior performance?

4.1 Dataset

A large set of PHP samples is collected and synthesized to evaluate the SmartEagleEye system. In the dataset, malicious samples mainly come from several famous open-source Webshell collection projects on GitHub※. Other samples are provided in collaboration with the security enterprise. Some of these samples are later proven to be not malicious; for example, simply file uploading or exploitation of certain vulnerabilities. These samples are excluded from the dataset. Normal samples in the dataset are randomly selected from the top-9 stared PHP projects on GitHub. All the duplicated samples, as well as those with irreparable compilation errors, are eliminated. Overall, the current version of the dataset includes 2553 malicious samples and 2635 normal ones.

4.2 Evaluation metrics

These widely used metrics are adopted to evaluate the effectiveness of different detection systems on the proposed task.

(1) Accuracy is the proportion of the samples that are correctly classified to all samples that are tested,

$$\text{Accuracy} = \frac{\text{TP}+\text{TN}}{\text{TP}+\text{TN}+\text{FP}+\text{FN}} \tag{18}$$

where TP denotes true positive, FP denotes false positive, TN denotes true negative, and FN denotes false negative.

(2) FPR is the ratio between the number of false positive and negative samples,

$$\text{FPR} = \frac{\text{FP}}{\text{FP}+\text{TN}} \tag{19}$$

(3) Precision is the number of true positive samples divided by the number of samples predicted to be positive. This metric measures the accuracy of some malicious samples determined by the system,

$$\text{Precision} = \frac{\text{TP}}{\text{TP}+\text{FP}} \tag{20}$$

(4) Recall measures the ratio of true positive samples to all the samples that are truly positive,

$$\text{Recall} = \frac{\text{TP}}{\text{TP}+\text{FN}} \tag{21}$$

(5) F1-score is the harmonic mean of precision and recall,

$$\text{F1-score} = \frac{2\times\text{Precision}\times\text{Recall}}{\text{Precision}+\text{Recall}} \tag{22}$$

4.3 Experiment I

First, 12 sets of controlled experiments use Term Frequency-Inverse Document Frequency (TF-IDF) as the word vector extraction method to investigate multiple text conversion methods, and the experiments involve four models: Gaussian Naive Bayes (GNB), Logistic Regression (LR), K-Nearest Neighbor (KNN), and Support Vector Machine (SVM). The experimental results prove that the effect of using OPCODE is the best, followed by AST, and the source code is the worst. The results are shown in Table 4.

Only the necessary structural components in the grammar are obtained by converting the source code to the AST to extract the source code information accurately. The OPCODE is compiled on the basis of the AST. Converting the source code into OPCODE can convert the complex and changeable source code containing substantial redundant information (such as comment content) into a limited number of OPCODE operating instructions. Using OPCODE helps quickly locate the execution of sensitive functions and parameter calls in the PHP code, thereby improving the accuracy of Webshell detection.

4.4 Experiment II

This experiment is designed to prove the effectiveness of the SmartEagleEye on the dataset. Evaluation objects include the gray-box and neural network

※https://github.com/tennc/webshell; https://github.com/ysrc/webshell-sample/tree/master/php

Table 4 Comparison between different text conversion methods.

(%)

Model	Accuracy	Recall	F1-score	Precision
Source code + TF-IDF + GNB	74.30	48.84	64.82	96.35
Source code + TF-IDF + LR	68.40	49.76	76.36	61.76
Source code + TF-IDF + KNN	51.22	51.32	67.67	51.14
Source code + TF-IDF + SVM	74.56	48.94	80.61	67.51
AST + TF-IDF + GNB	79.91	59.49	74.17	98.93
AST + TF-IDF + LR	76.75	55.06	70.75	98.93
AST + TF-IDF + KNN	67.02	37.50	53.73	94.74
AST + TF-IDF + SVM	77.88	59.83	74.09	97.28
OPCODE + TF-IDF + GNB	87.62	77.89	86.37	96.90
OPCODE + TF-IDF + LR	98.22	98.66	98.24	97.82
OPCODE + TF-IDF + KNN	97.69	97.22	97.69	98.16
OPCODE + TF-IDF + SVM	98.55	99.14	98.57	98.01

analyzers. Quick detection on agent is excluded in this experiment because its implementation is similar to traditional open-source detection tools, which will be used as the comparison baseline in Experiment II.

First, the hidden sizes of Bi-LSTM are set to 128 based on the general architecture of Conv-BiLSTM described in Section 3.4. Other hyperparameters are set to the following default values: the batch size is set to 32, the learning rate is set to 0.001, and the number of epochs is set to 10. The dataset is randomly divided into two parts: 70% of the samples are used for training, and the remaining 30% are used for testing. Convergence results are shown in Fig. 12, the blue curve in the graph represents the change in loss with each epoch, while the yellow curve represents the change in accuracy with each epoch.

The Conv-BiLSTM model is deployed in the neural network analyzer upon completion of training, and SmartEagleEye is evaluated. The results are shown in Table 5, with metrics provided in Section 4.2. SmartEagleEye identifies all 2553 malicious samples and incorrectly labels ten normal samples as malicious.

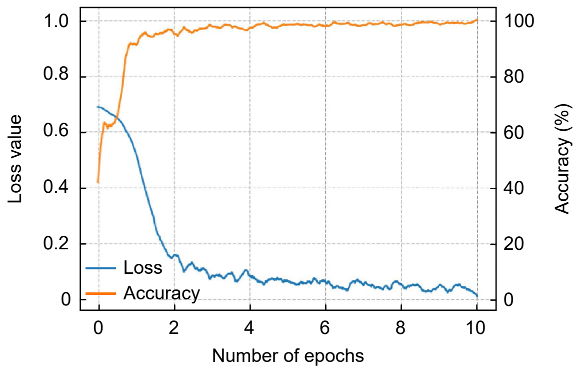


Fig. 12 Training process.

Afterward, OPCODE is used as input, and BERT is utilized for word vector extraction using CNN, LSTM, and Conv-BiLSTM for experiments. Compared to directly using LSTM or CNN, Conv-BiLSTM has significantly improved recall, F1-score, and accuracy. The results are shown in Table 6. Conv-BiLSTM extracts the feature information of OPCODE from a deep perspective and comprehensively extracts the feature information of the OPCODE text, thereby obtaining good detection results.

Table 5 Evaluation of Conv-BiLSTM.

TP	FP	TN	FN	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)
2549	13	2622	4	99.67	99.49	99.66	99.84

Table 6 Comparison among different models.

(%)

Model	Accuracy	Recall	F1-score	Precision
OPCODE + BERT + CNN	99.24	98.99	99.07	99.19
OPCODE + BERT + LSTM	97.04	98.71	97.10	95.75
Conv-BiLSTM	**99.67**	**99.49**	**99.66**	**99.84**

4.5 Experiment III

In this experiment, SmartEagleEye (gray-box and neural network analyzers) is compared with the following existing Webshell detection systems.

• Windows Defender (WinDefender): Built-in antivirus software in Microsoft Windows systems.

• Tencent PC Manager (QQPCMgr): The fastest growing general-purpose security software in mainland China.

• WebShell.Pub (HemaAV): New multi-engine Webshell detection system based on machine learning, cloud technology, and big data technology.

• findWebShell: An open-source Webshell detection tool.

• Shell-Detector: An open-source Webshell detection tool designed for PHP.

• php-malware-finder: An open-source Webshell detection tool designed for PHP.

• D-Shield: IIS designed for active defense protection software to prevent websites and servers from intrusion.

• 360 Security: Cloud security antivirus software integrated with the mainstream killing engine.

• HUORONG SECURITY: A terminal security management system.

The dataset in Section 4.1 and the metrics in Section 4.2 are used to compare these detection systems, and the results are shown in Table 7. This experiment is inconsistent with the design and application scenarios of SmartEagleEye. The FPR in the production environment should be substantially lower than that in this experiment. However, this experiment can reflect the FPR of SmartEagleEye in extreme scenarios.

Compared with the existing detection systems, SmartEagleEye still demonstrates significant advantages, even in the worst scenario. The widely used commercial security products perform poorly in Webshell recognition mainly because such products are not designed for servers and are not optimized for Webshell. Most use simple fingerprint-matching methods and static analysis techniques for detection. Compared with the EagleEye system, these engines lack the key behavioral information introduced by dynamic execution and the generalization capability of machine learning. Thus, the detection rate lags behind the EagleEye system. Some scan engines mark files as suspicious and issue an alert, resulting in a phenomenal increase in FPR while achieving relatively good detection rates for such machines.

4.6 Summary

Models in Experiments I and II are widely used in existing works, and systems in Experiment III are the mainstream tools currently utilized in the industry. Experiments have proven that SmartEagleEye has shown better performance compared to machine learning-based or industrial approaches.

5 Conclusion and Future Work

This paper introduces the SmartEagleEye system, a cloud-based, client-server distributed architecture for PHP-based Webshell detection, to improve the security of the cloud environment. The two core modules of this system, the gray-box and neural network analyzers, work jointly to complete detection tasks. A series of evaluation experiments are designed for the SmartEagleEye and some baseline methods. SmartEagleEye achieves more than 99% accuracy on

Table 7 Comparison among different detection systems.

(%)

Detector	Accuracy	FPR	Precision	Recall	F1-score
SmartEagleEye	**99.29**	**1.40**	**98.60**	**100.00**	**99.28**
WinDefender	62.20	0.00	100.00	23.30	37.80
QQPCMgr	60.50	0.00	100.00	19.60	32.80
HemaAV*	83.10	1.30	98.10	67.00	79.60
findWebShell	59.60	0.40	97.70	18.30	30.90
Shell-Detector*	81.40	8.00	89.50	70.40	78.80
php-malware-finder	62.00	0.80	76.80	23.20	37.60
D-Shield	92.44	10.37	95.18	90.44	92.75
360 Security	63.16	74.85	100.00	57.96	73.38
HUORONG SECURITY	73.92	52.99	100.00	66.07	79.57

Note: * denotes samples are labeled as suspicious included.

different datasets overall and acceptable false positives in the meantime. Experiments show that the proposed system outperforms other recognized competitive products.

The current version of the SmartEagleEye system has been proved effective on the dataset but can still be improved. Considering the gray-box analyzer, additional built-in functions commonly related to Webshells will be obtained and added to a dangerous function list. Moreover, support for the detection of object-oriented Webshells, which are not currently available, will be provided. Current sample preprocessing has some weaknesses when dealing with complex samples. Thus, the implementation of sample preprocessing must be improved.

Further efforts can also be made to improve the neural network module. Properly representing and modeling the Source code remains challenging. Converting PHP code to OPCODE sequence first is chosen in this paper, representing code as OPCODE tokens. This approach may intuitively be feasible with natural language or other sequences with an autoregressive nature. However, capturing the long-range dependencies between the same variables or function names in distant locations is difficult. A superior solution for learning tasks in Source code remains a popular area of research. Some researchers proposed using highly structural intermediates to represent code, such as trees and graphs[16–18]. Correspondingly, models designed for learning structural data, such as graph neural networks, are adopted to replace sequential ones[19].

Moreover, the Webshell detection task is regarded as a classification problem. This approach naturally has a demand for labeled data to train models. However, labeled data that fits the target task is not always easily available, especially in code learning problems. Labels for codes can only be acquired by experts in this field; thus, labeling can also be a labor-intensive task. An alternative approach is to store a database of interesting samples. Considering an under-detection sample, the entire database is searched, and the similarity between this sample and that in the database is computed. Hence, malicious codes could ideally be recognized by at least one known sample of the type that exists in the database[20]. This matching method possibly performs effectively under insufficient or expensive to obtain training data[21]. The exploration of these improvements will be addressed in future work.

6 Related Work

Webshell may incur a heavy loss. Thus, even if the ideal approach is to prevent breaches in web applications as well as exploitation, researchers in the domain of network security have focused on some conventional protective countermeasures[22–25]. Accurately spotting and recognizing Webshells once uploaded on the server are still beneficial. However, accurate recognition is never a trivial task due to the complex nature of the web environment and the massiveness of web applications.

Researchers have designed two kinds of approaches, namely static and dynamic, to detect Webshells. However, static and dynamic countermeasures inherently possess advantages and disadvantages. Traditional static methods, including signature or hash-based fingerprint matching, are widely used to build the sample database. These methods are naturally deficient in the face of obfuscated or brand-new Webshell variations in the wildness of the Internet. Other methods, such as static code analysis, usually introduce an intensive cost of design and labor implementation[26–28]. Dynamic solutions are effective for capturing the actual behavior of malicious codes. However, these solutions must execute the code first to obtain execution traces, introducing code coverage problems and system overhead. NeoPi[29], known as one of the most popular Webshell detection tools, is based on statistical methods and uses the following five methods: entropy, longest word, index of coincidence, signature, and compression. Since it will not be updated after 2015, its feature library is relatively old. Ying and Yong[30] used Apriori and FP-Growth algorithms for Webshell detection based on NeoPi and improved accuracy. Le et al.[31] proposed a solution that combines protection and scalability. The solution chooses taint analysis and pattern matching as the primary methods for Webshell detection.

From another perspective, Webshell detection is regarded as a classification problem that aims to determine whether a piece of code belongs to the malicious or benign class. Machine learning approaches, especially deep learning, have been applied in various fields and achieved remarkable success[32]. Machine learning shows excellent potential in classification tasks with a large amount of training data (for example, image classification and audio recognition)[33–35]. Intuitively, the prevalence of

Webshells provides an excellent chance to learn from real-world samples. Cui et al.[36] proposed a detection model that uses the OPCODE sequence generated during the PHP script execution as the feature vector and then employs the random forest algorithm to classify the PHP files. FRF-WD[37] is a PHP Webshell detection model based on the combination of fastText and random forest algorithms. Zhang et al.[38] used the URL and request body part of the HTTP network request as the original data source of the model. They directly used the CNN + LSTM network to extract and distinguish the entire content, and established a complete end-to-end deep neural network for Webshell flow detection. Zhou et al.[39] utilized a deep learning technique called LSTM Recurrent Neural Networks (RNNs), to detect Webshells, and their results indicate that the single-layer model may demonstrate the best performance on Webshell detection. The proposed approach of Gogoi et al.[40] analyzes the function call and the use of super global variables commonly employed in PHP Webshells using a deep learning technique. Qi et al.[41] designed a generic static end-to-end detection framework with a deep neural network for Webshell, free from human labor and domain knowledge. Betarte et al.[42] introduced a combination of a classifier and an *n*-gram model to improve the detection capabilities of WAF application firewalls. Nataraj et al.[43] indicated that if a process converted malware binaries into grayscale images, then the computer could recognize malware from different families according to their layout and texture. The idea in Ref. [44] combines the image identification capability of CNN with malware detection tasks. Another line of deep learning focuses on the textual nature of codes[45]; thus, RNN is used to model code sequences[12]. The current work typically falls into this category.

Acknowledgment

Special thanks to Prof. Binbin Yong for his help with this work. This work was supported by the National Key R&D Program of China (No. 2020YFC0832500), the Science and Technology Plan of Gansu Province (Nos. 22ZD6GA048 and 22YF7GA004), and the Supercomputing Center of Lanzhou University.

References

[1] The NIST definition of Cloud computing, https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-145.pdf, 2010.

[2] A. K. Sandhu, Big data with cloud computing: Discussions and challenges, *Big Data Mining and Analytics*, vol. 5, no. 1, pp. 32–40, 2022.

[3] M. Azrour, J. Mabrouki, A. Guezzaz, and Y. Farhaoui, New enhanced authentication protocol for internet of things, *Big Data Mining and Analytics*, vol. 4, no. 1, pp. 1–9, 2021.

[4] Web shell, https://en.wikipedia.org/wiki/Webshell, 2023.

[5] Acunetix Web Application Vulnerability Report 2019, https://www.acunetix.com/acunetix-web-application-vulnerability-report/, 2020.

[6] Zend Engine 2 Opcodes, https://php-legacy-docs.zend.com/manual/php5/en/internals2.opcodes, 2022.

[7] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, BERT: Pre-training of deep bidirectional transformers for language understanding, in *Proc. 2019 Conf. North American Chapter of the Association for Computational Linguistics*: *Human Language Technologies, Volume 1* (*Long and Short Papers*), Minneapolis, MN, USA, 2018, pp. 4171–4186.

[8] 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 *Proc. 25th Int. Conf. World Wide Web*, Montréal, Canada, 2016, pp. 1021–1032.

[9] Usage statistics of PHP for websites, https://w3techs.com/technologies/details/pl-php, 2020.

[10] AMPQ Homepage, https://www.amqp.org/, 2022.

[11] PSR-12: Extended Coding Style, https://www.php-fig.org/psr/psr-12/, 2020.

[12] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, VulDeePecker: A deep learning-based system for vulnerability detection, in *Proc. Network and Distributed System Security Symp.*, San Diego, CA, USA, 2018, pp. 1–15.

[13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, Attention is all you need, arXiv preprint arXiv:1706.03762, 2023.

[14] Activation function-wikipedia, https://en.wikipedia.org/wiki/Activation_function, 2023.

[15] J. F. Kolen and S. C. Kremer, Gradient flow in recurrent nets: The difficulty of learning long-term dependencies, in *A Field Guide to Dynamical Recurrent Networks*. Los Alamitos, MX, USA: Wiley-IEEE Press, 2001, pp. 237–243.

[16] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, code2vec: Learning distributed representations of code, arXiv preprint arXiv:1803.09473, 2018.

[17] U. Alon, S. Brody, O. Levy, and E. Yahav, code2seq: Generating sequences from structured representations of code, arXiv preprint arXiv:1808.01400, 2019.

[18] M. Allamanis, M. Brockschmidt, and M. Khademi, Learning to represent programs with graphs, arXiv preprint arXiv:1711.00740, 2018.

[19] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, A comprehensive survey on graph neural networks, arXiv preprint arXiv:1901.00596, 2019.

[20] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, VulPecker:

An automated vulnerability detection system based on code similarity analysis, in *Proc. 32nd Annu. Conf. Computer Security Applications*, Los Angeles, CA, USA, 2016, pp. 201–213.

[21] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, Graph matching networks for learning the similarity of graph structured objects, arXiv preprint arXiv:1904.12787, 2019.

[22] B. A. Jnr, Managing digital transformation of smart cities through enterprise architecture–a review and research agenda, *Enterp. Inf. Syst.*, vol. 15, no. 3, pp. 299–331, 2021.

[23] W. Tan, Y. Zhao, X. Hu, L. Xu, A. Tang, and T. Wang, A method towards Web service combination for cross-organisational business process using QoS and cluster, *Enterp. Inf. Syst.*, vol. 13, no. 5, pp. 631–649, 2019.

[24] K. Alieyan, A. Almomani, M. Anbar, M. Alauthman, R. Abdullah, and B. B. Gupta, DNS rule-based schema to botnet detection, *Enterp. Inf. Syst.*, vol. 15, no. 4, pp. 545–564, 2021.

[25] A. Dahiya and B. B. Gupta, A PBNM and economic incentive-based defensive mechanism against DDoS attacks, *Enterp. Inf. Syst.*, vol. 16, no. 3, pp. 406–426, 2022.

[26] J. Dahse and T. Holz, Simulation of built-in PHP features for precise static code analysis, in *NDSS'14*, San Diego, CA, USA, 2014, pp. 23–26.

[27] N. Jovanovic, C. Kruegel, and E. Kirda, Pixy: A static analysis tool for detecting web application vulnerabilities, in *Proc. 2006 IEEE Symp. Security and Privacy*, Berkeley/Oakland, CA, USA, 2006, pp. 258–263.

[28] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna, Saner: Composing static and dynamic analysis to validate sanitization in web applications, in *Proc. 2008 IEEE Symp. Security and Privacy*, Oakland, CA, USA, 2008, pp. 387–401.

[29] NeoPI, https://github.com/CiscoCXSecurity/NeoPI, 2023.

[30] Z. Ying and H. Yong, Webshell detection method based on correlation analysis, *Journal of Information Security Research*, vol. 4, no. 3, p. 5, 2018.

[31] V. G. Le, H. T. Nguyen, D. N. Lu, and N. H. Nguyen, A solution for automatically malicious web shell and web application vulnerability detection, in *Proc. 8th Int. Conf. Computational Collective Intelligence*, Halkidiki, Greece, 2016, pp. 367–378.

[32] W. Zhong, N. Yu, and C. Ai, Applying big data based deep learning system to intrusion detection, *Big Data Mining and Analytics*, vol. 3, no. 3, pp. 181–195, 2020.

[33] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. R. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, et al., Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups, *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 82–97, 2012.

[34] A. Krizhevsky, I. Sutskever, and G. E. Hinton, ImageNet classification with deep convolutional neural networks, *Commun. ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[35] J. Pennington, R. Socher, and C. Manning, Glove: Global vectors for word representation, in *Proc. 2014 Conf. Empirical Methods in Natural Language Processing* (*EMNLP*), Doha, Qatar, 2014, pp. 1532–1543.

[36] H. Cui, D. Huang, F. Yong, L. Liang, and H. Cheng, Webshell detection based on random forest-gradient boosting decision tree algorithm, in *Proc. 2018 IEEE Third Int. Conf. Data Science in Cyberspace* (*DSC*), Guangzhou, China, 2018, pp. 153–160.

[37] Y. Fang, Y. Qiu, L. Liu, and C. Huang, Detecting webshell based on random forest with FastText, in *Proc. 2018 Int. Conf. Computing and Artificial Intelligence*, Chengdu, China, 2018, pp. 52–56.

[38] H. Zhang, H. Guan, H. Yan, W. Li, Y. Yu, H. Zhou, and X. Zeng, Webshell traffic detection with character-level features based on deep learning, *IEEE Access*, vol. 6, pp. 75268–75277, 2018.

[39] Z. Zhou, L. Li, and X. Zhao, Webshell detection technology based on deep learning, in *Proc. 2021 7th IEEE Int. Conf. Big Data Security on Cloud* (*BigDataSecurity*), *IEEE Int. Conf. High Performance and Smart Computing* (*HPSC*), *and IEEE Int. Conf. Intelligent Data and Security* (*IDS*), New York, NY, USA, 2021, pp. 52–56.

[40] B. Gogoi, T. Ahmed, and R. G. Dinda, PHP web shell detection through static analysis of AST using LSTM based deep learning, in *Proc. 2022 First Int. Conf. Artificial Intelligence Trends and Pattern Recognition* (*ICAITPR*), Hyderabad, India, 2022, pp. 1–6.

[41] L. Qi, R. Kong, Y. Lu, and H. Zhuang, An end-to-end detection method for WebShell with deep learning, in *Proc. 2018 Eighth Int. Conf. Instrumentation & Measurement*, *Computer*, *Communication and Control* (*IMCCC*), Harbin, China, 2018, pp. 660–665.

[42] G. Betarte, E. Giménez, R. Martínez, and Á. Pardo, Machine learning-assisted virtual patching of web applications, arXiv preprint arXiv:1803.05529, 2018.

[43] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath, Malware images: Visualization and automatic classification, in *Proc. 8th Int. Symp. Visualization for Cyber Security*, Pittsburgh, PA, USA, 2011, p. 4.

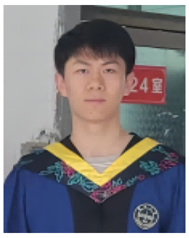
[44] J. Lin, G. Sun, J. Shen, D. E. Pritchard, P. Yu, T. Cui, D. Xu, L. Li, and G. Beydoun, From computer vision to short text understanding: Applying similar approaches into different disciplines, *Intelligent and Converged Networks*, vol. 3, no. 2, pp. 161–172, 2022.

[45] Q. Zhu, X. Ma, and X. Li, Statistical learning for semantic parsing: A survey, *Big Data Mining and Analytics*, vol. 2, no. 4, pp. 217–239, 2019.

Xin Liu received BEng and MEng degrees in computer science from Lanzhou University, China in 2017 and 2019, respectively. He is currently a PhD candidate at Lanzhou University, China. His research interests include code analysis, blockchain security, and open source intelligence.

Yingli Zhang received the BEng degree in information security from Hunan University, China in 2019. She is currently a PhD candidate at Lanzhou University, China. Her research interests include blockchain security, IoT security, and machine learning.

Qingchen Yu received the BEng degree in computer science from Lanzhou University, China in 2017. She is a PhD candidate at College of Computer Science and Technology, Zhejiang University, China. Her research interests include intersection of deep learning, program analysis, and compiler optimization.



Jiajun Min received the BEng degree from Institute of Disaster Prevention, China in 2018. He is currently a master student at Lanzhou University, China. His research interests include intersection of deep learning and web security.

Jun Shen received the PhD degree in computer science from Southeast University, China in 2001. He is currently an associate professor at School of Computing and Information Technology, University of Wollongong, Wollongong, NSW, Australia, where he has been the head of Postgraduate Studies and the chair of the School Research Committee since 2014. His research interests include computational intelligence, bioinformatics, cloud computing, and learning technologies (including massive open online courses).

Rui Zhou received the PhD degree in applied mathematics from Lanzhou University, China in 2010. He is currently an associate professor and working at School of Information Science and Engineering, Lanzhou University, China. His main research interests include distributed systems embedded systems and machine learning.

Qingguo Zhou received the BS and MS degrees in physics from Lanzhou University, China in 1996 and 2001, respectively, and the PhD degree in theoretical physics from Lanzhou University, China in 2005. Currently, he is a professor at Lanzhou University, China. His research interests include safety-critical systems, embedded systems, and real-time systems.