applied sciences

MDPI

Article

WTA: A Static Taint Analysis Framework for PHP Webshell

Jiazhen Zhao [1,2], Yuliang Lu [1,2,*], Xin Wang [3], Kailong Zhu [1,2] and Lu Yu [1,2]

[1] College of Electronic Engineering, National University of Defense Technology, Hefei 230037, China; jiazhenzhao@nudt.edu.cn (J.Z.); zhukailong@nudt.edu.cn (K.Z.); yulu@nudt.edu.cn (L.Y.)
[2] Anhui Province Key Laboratory of Cyberspace Security Situation Awareness and Evaluation, Hefei 230037, China
[3] School of Computer Science and Engineering, Central South University, Changsha 410083, China; 0906170214@csu.edu.cn
* Correspondence: luyuliang@nudt.edu.cn

Abstract: Webshells are a malicious scripts that can remotely control a webserver to execute arbitrary commands, steal sensitive files, and further invade the internal network. Existing webshell detection methods, such as using pattern matching for webshell detection, can be easily bypassed by attackers using the file include and user-defined functions. Furthermore, detecting unknown webshells has always been a problem in the field of webshell detection. In this paper, we propose a static webshell detection method based on taint analysis, which realizes accurate taint analysis based on ZendVM. We first converted the PHP code into Opline sequences, analyzed the Opline sequences in order, and marked the externally imported taint source. Then, the propagation of the taint variables was tracked, and the interprocedural analysis of the taint variables was performed. Finally, considering the dangerous functions' call and the referencing of the taint variables at the point of the taint sink, we completed the webshell judgment. Based on this method, we constructed a taint analysis prototype system named WTA and evaluated it with a benchmark dataset by comparing its performance with popular webshell detection tools. The results showed that our method supports interprocedural analysis and has the ability to detect unknown webshells and that WTA's performance surpasses well-known webshell detection tools such as D-shield, SHELLPUB, WebshellKiller, CloudWalker, ClamAV, LoKi, and findbot.pl.

Keywords: webshell; Opline; taint analysis; webshell detection; WTA

Citation: Zhao, J.; Lu, Y.; Wang, X.; Zhu, K.; Yu, L. WTA: A Static Taint Analysis Framework for PHP Webshell. *Appl. Sci.* **2021**, *11*, 7763. https://doi.org/10.3390/app11167763

Academic Editor: Juan-Carlos Cano

Received: 29 May 2021
Accepted: 19 August 2021
Published: 23 August 2021

1. Introduction

With the rapid development of network technology, web applications [1] have become the dominant form by which Internet companies provide users with web services. At the same time, all kinds of network attacks on web applications have become the main problem threatening Internet security. In February 2020, Microsoft released a report, *Microsoft Defender Advanced Threat Protection* [2], showing that it detects approximately 77,000 active webshells [3] per day, which means that webshells have become some of the most popular types of malware today. Webshells are a malicious network backdoor that can exist in multiple scripting languages [4], allowing attackers to gain system privileges or control the webserver by executing arbitrary commands [5]. Attackers can use webshells to carry out a series of malicious operations, such as accessing server databases and sensitive files, stealing and tampering with user data, modifying the home page of a website, and so on. In terms of website security, it is crucial to detect webshell files and delete them [6].

According to the scripting language, webshells can mainly be divided into three types, namely ASP, PHP, and JSP scripting Trojans [7]. Due to its simple syntax and high development efficiency, PHP has become the first choice for developing various types of web applications [8]. Therefore, this paper mainly studies the PHP webshell detection method.

At present, webshell detection methods can be divided into dynamic feature detection and static feature detection.

The dynamic feature detection method is based on the characteristics of the webshell execution process, such as the behaviors of webshell files, webshell communication traffic [9], and other characteristics [7]. This method only works when the webshell is executing dynamically. On the one hand, this method has a certain ability to detect new variants of scripts and is good at detecting webshell features generated by operations [10]. On the other hand, this method must detect the traffic during the operation and communication process and needs to maintain a large behavioral characteristic library, so it may consume most of the computing resources of the server.

The static feature detection method is mainly based on the text content of webshell and network log information [11,12] for analysis and detection. Regular expressions [13] were the earliest method used for webshell content detection. Its disadvantage is that it can only extract features from the existing known webshells, and it needs to be constantly updated [14]. D-Shield [15] is a currently popular static webshell detection tool. It uses signature database matching to detect webshells and divides webshells into six levels according to the degree of damage: Level 0 is not a webshell, and Level 6 is a known webshell. Therefore, static feature detection methods cannot detect unknown or new webshells. In addition, due to the constant evolution and iteration of code obfuscation and code encryption techniques, webshells can easily bypass regular methods, which are based on regular expressions. Moreover, the static feature detection method has no way to conduct interprocedural analysis, that is to detect the included files and user-defined dangerous function, so the detection method is based on the feature code and syntax analysis, and the dangerous function [16] name matching can be easily bypassed.

In recent decades, the role of taint analysis [17] in program analysis has attracted extensive attention from researchers. Static taint propagation analysis, also called static taint analysis [18], is the analysis of the data dependencies between variables to test whether data can be propagated from the taint source to a point of the taint sink without running or modifying them. The object of static taint analysis is generally the source code or intermediate representation of the program. RIPS [19] uses taint analysis to analyze PHP codes in a static way, which is based on AST derived from syntax analysis. Yu Li et al. [20] proposed a detection platform named Shellbreaker. They extracted eight new source codes and AST syntactic and semantic features. Two of the features are *explicit data flow* and *implicit data flow*, and they are extracted by taint analysis. Then, the eight features are fused into a vector. Finally, a statistical classifier is used to analyze the feature vector. However, this method has a limited detection effect on one-sentence webshells [21], because several types of features extracted by this method are aimed at self-adaptive webshells.

The performance of the dynamic feature detection method is poor, and the construction of the environment is complex. Traditional static feature detection methods have difficulties in detecting unknown webshells and lack the capability to perform interprocedural analysis. In addition, there has been some research that has used taint analysis for webshell detection with limited effect. To address the above challenges, this paper proposes a webshell detection method based on static taint analysis.

The main contributions of this paper are as follows:

(1) We applied the ZendVM instruction set to the field of taint analysis for the first time and defined the taint propagation rules and taint sink rules of the instruction set;
(2) We proposed a novel static detection method based on taint analysis for PHP webshells. The method can carry out interprocedural analysis and detect more unknown webshells;
(3) We implemented a taint analysis prototype system named WTA for webshell detection and evaluated the effectiveness of our method by comparing it with existing tools through a benchmark dataset consisting of ten webshell datasets and six CMSs.

The remainder of this paper is organized as follows. Section 2 describes the background information on PHP. Section 3 introduces the include-type webshell, the user-defined function-type webshell, and the unknown webshell, which bring challenges to

webshell detection. The overview of the proposed approach is described in Section 4. Section 5 describes the details of the three key steps in our method. Section 6 evaluates our method. We summarize the related work in Section 7 and provide our conclusions in Section 8.

2. Background

PHP [22]. PHP is a popular scripting language that is particularly suited to web development. It runs in four modes: PHP-CLI, PHP-CGI, PHP-FPM, and PHP-MOD. PHP has three main characteristics: First, PHP code is open-source, and the community is active, so the number of people using PHP is large. Second, the syntax of PHP is simple; process-oriented and object-oriented programming can be mixed; it is easy to use; it has many built-in modules. Third, PHP has strong expansibility. In the process of the continuous development of PHP, it can take into account the performance and the current popular frameworks and has a good extension interface for developers to use.

ZendVM [23]. The virtual machine of a programming language is a program that can run an intermediate language. The intermediate language is an abstract set of instructions compiled from the native language and is the input of the virtual machine during its execution. The virtual machine of the PHP language is called the Zend Virtual Machine (ZendVM). The ZendVM will perform lexical analysis and syntactic analysis on the target PHP file to generate the AST, then compile the AST into Opcodes, and finally, execute the Opcodes and output the results. The workflow diagram of the ZendVM is shown in Figure 1.

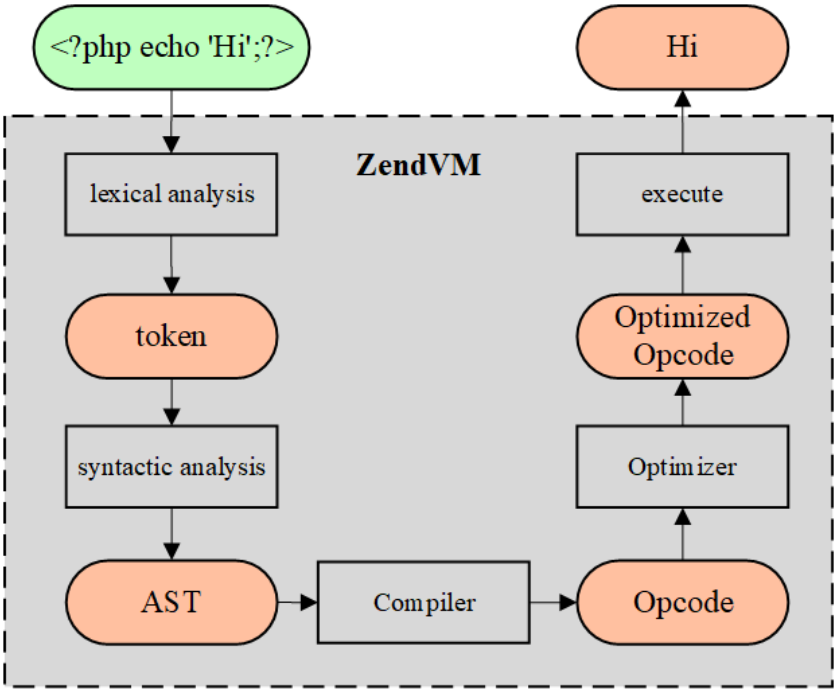

Figure 1. The workflow of the ZendVM.

Opline and Opcode [24]. The ZendVM's instruction is called the **Opline**, and each instruction corresponds to an **Opcode**. Oplines are generated after the compilation of the PHP codes. The ZendVM executes PHP codes according to different Oplines. The Opline consists of operation instructions and operands and returns the value, which is similar to a machine instruction. The corresponding structure of the Opline is zend_op. The basic information of the zend_op structure is shown in Listing 1.

PHP extensions [25]. As mentioned above, one of the main reasons for the popularity of PHP is that a large number of extensions are available. Whatever the needs of web developers are, they are likely to find them addressed by the releases of PHP. The releases

of PHP include many extensions that support a variety of databases, graphical file formats, compression, and XML technology. Web developers can be involved in the PHP compilation phase and redefine the PHP compilation functions for deeper operations by writing PHP extensions.

Listing 1. Basic information of the structure zend_op.

```
struct _zend_op {
        const void *handler;     // The handler called when
            Opcodes are executed
        znode_op op1;            // Operand1
        znode_op op2;            // Operand2
        znode_op result;         // Used to hold the result after
            the execution
        uint32_t extended_value;// Saves some additional
            information
        uint32_t lineno;         // Saves the line of the source
            code
        zend_uchar opcode;       // The Opcode is used
        zend_uchar op1_type;
        zend_uchar op2_type;
        zend_uchar result_type;
};
```

Vulcan Logic Dumper (VLD) [26]. VLD is a PHP extension that outputs Oplines by hook. By using the VLD, developers can view the Oplines of the target PHP codes, allowing them to gain a deeper understanding of the PHP codes.

3. Motivation

The lack of the ability to perform interprocedural analysis and detect unknown webshells is the main challenge in the field of webshell detection. We present two examples to illustrate the significance of interprocedural analysis and the difficulty of unknown webshell detection.

Interprocedural analysis [27]. There are two types of webshells that require interprocedural analysis. The first is **include-type webshells**, and the second is **user-defined function-type webshells**. Include-type webshells refer to an attacker who puts the body of the webshell into a text file, image file, or any file in other formats. For example, the attacker puts the body of the webshell into hello.txt, while the webshell file (attack.php) has one "include" statement used to include the file hello.txt. Therefore, if the webshell detection tool only scans the file attack.php, without an in-depth analysis of the contents of the file hello.txt contained in attack.php, the webshell can bypass the detection.

Listing 2 shows two include-type webshell examples called include-webshell-1 and include-webshell-2. Listing 3 shows the file hello.txt included in an include-type webshell. In fact, **include-webshell-1** and **include-webshell-2** have the same function: they all include the webshell body hello.txt. However, detecting the two files using the popular tool D-Shield [15] obtains different results. The detection results indicate that **include-webshell-1** is a webshell of Level 3, and the reason is "suspicious include". D-Shield considers **include-webshell-2** not to be a webshell. From this experiment, we can see that the tool D-Shield simply uses **include 'filename'** as a matching pattern and considers the file to be a webshell once the match is found, while include-webshell-2 bypasses detection by replacing spaces with parentheses. This also indicates that the tool does not detect the contents of the include file and cannot detect the include-type webshells, which will be further explained in Experiment 1 of Section 6.

Listing 2. Include-type webshells.

```
<?php
include 'hello.txt';
?>
(a) include-webshell-1

<?php
include('hello.txt');
?>
(b) include-webshell-2
```

Listing 3. hello.txt.

```
<?php
if(md5($_GET['pass'])=='21232f297a57a5a743894a0e4a801fc3')
{
        eval($_POST[console]);
}
else
{
        die('exit');
}
?>
```

User-defined function-type webshells refer to the way that attackers bypass the scanning of known dangerous functions by creating user-defined functions and executing system commands in the user-defined functions.

Listing 4 shows a user-defined function webshell. Two user-defined functions, **dynamic** and **newassert**, are included in the sample to obtain the taint source and to call the dangerous function assert(). The sample is tested by D-Shield [15], and the test results show that the sample is a webshell of Danger Level 1 (a webshell of Danger Level 1 can be considered as a normal file). The reason is that the variable $c is used in this file. Therefore, D-Shield cannot actually detect user-defined function webshells, which will be further explained in Experiment 1 in Section 6.

Listing 4. User-defined function-type webshell.

```
<?php
function dynamic(){
        $a = "%7U!/4U0'␣'";
        $a = convert_uudecode($a);
        return $a;
}
function newassert(){
        $c = "&87-S97)T␣'";
        $c = convert_uudecode($c);
        return $c;
}
$a = dynamic();
$b = $$a;
$c = newassert();
$c ($b['x']);$
```

Unknown webshell [28]. An unknown webshell is a webshell that has not yet been discovered. Since such webshells are not captured, current webshell detection tools and antivirus software do not have corresponding sample signatures and cannot detect unknown webshells. Most of the latest methods are based on malicious pattern matching, such as the tool D-Shield, in which keywords are usually defined by domain experts. Therefore, the detection effect depends on the experts, and new webshells are difficult to detect. In addition, there are many research works on webshell detection methods based on machine learning and neural networks, such as cnn-webshell [9] and Yong e al.'s work [10], whose essence is to extract the features of known webshells for analysis. Therefore, it the features

of unknown webshells differ greatly from those of known webshells, it will be difficult to detect the unknown webshells, which will be further explained in Experiment 2 in Section 6.

4. Overview

In order to solve the above limitations of the existing methods, we propose a webshell static detection method based on taint analysis. This method aims to improve the ability of the static detection of unknown webshells and provide the capability to perform interprocedural analysis.

The method proposed in this paper includes the following seven steps, as shown in Figure 2.

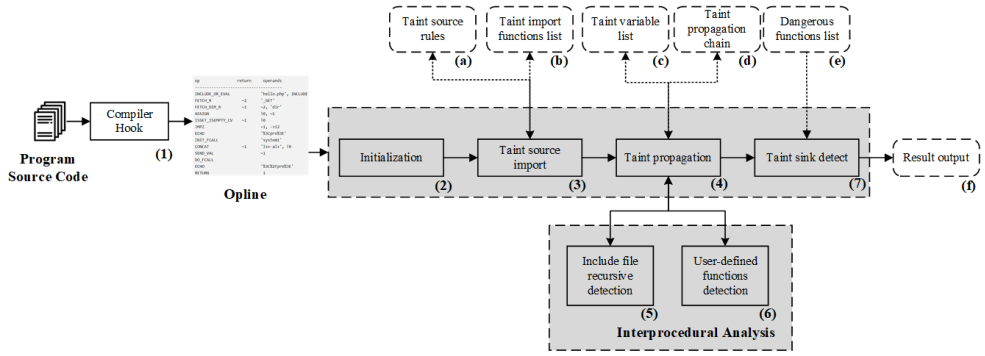

Figure 2. The overview of our method.

Compiler hook (1) obtains the intermediate code Opline sequences through the PHP Compiler function. The implementation of this module uses the codes of the VLD [26] extension for reference.

Initialization (2) completes the preparation work before the taint analysis. The preparation work is to initialize the data structures of Taint import functions list (b), Taint variable list (c), and Dangerous functions list (e). Then, Taint source rules (a), Taint import functions list (b), and Dangerous function list (e) also require the user to set the initial values. For example, the user fills in Dangerous functions list (e) with functions that can be used as the webshell dangerous functions from the PHP functions library [29], such as exec, shell_exec, system, and so on.

After Initialization (2), the taint analysis framework starts to analyze the Opline sequences in order, which are the outputs of (1). **Taint source import** (3) conducts the detection according to the preset Taint source rules (a) and Taint import functions (b). When the taint source is found to be imported, the variable used to store the taint source will be stored in Taint variable list (c), and a new linked list will be created with this variable as the header node. Then, the linked list will be saved in Taint propagation chain (d).

Taint propagation (4) analyzes the propagation path of the taint variable in the Opline sequences, adds the tainted variable to Taint variable list (c), and adds the new taint variables to the corresponding taint propagation chain according to the propagation path.

The interprocedural analysis of webshells is one of the contributions of this paper, which consists of two modules: Include file recursive detection (5) and User-defined functions detection (6).

Include file recursive detection (5) will start the taint analysis subprocess when it meets the "include" expression, perform recursive detection on the included files, and return the detection results to the main process through the message queue (IPC [30]).

User-defined functions detection (6) actually has the highest priority for execution. Compiler hook (1) first obtains the Opline sequences, which come from the user-defined functions in the target PHP codes, then it obtains the Opline sequences, which come from the user-defined functions of the user-defined class, and finally, it obtains the Opline sequences produced by the other parts. User-defined functions detection (6) performs

taint analysis for user-defined functions and regards the parameters of functions as the taint source. After taint propagation, the user-defined functions are defined as dangerous functions/taint import functions and added into Dangerous functions list (e)/Taint import functions list (b) once the parameters that include the taints are imported into the dangerous functions.

Taint sink detect (7) judges whether a dangerous function is called by the function call instruction according to Dangerous functions list (e).

When a dangerous function is called and the parameter of a dangerous function is a taint variable, it will be added into Taint propagation chain (d), which is marked as the webshell taint propagation chain. When the final result is output, the taint propagation chain is detected. If there is a webshell taint propagation chain, it will be presented in Result output (f).

There are several challenges that need to be solved to implement this architecture:

(1) The data structures of the taint variables list, taint import functions list, taint propagation chain, dangerous functions list;

(2) Taint propagation rules [31] of the PHP Opline;

(3) Taint sink rules of the PHP Opline.

These difficulties will be addressed in the next section.

5. Methodology

5.1. Data Structures of Auxiliary Lists

The data structures of auxiliary lists refer to the taint variables list, taint import functions list, taint propagation chain, and dangerous functions list in the initialization module, while the taint source rules are hard coded in the corresponding functions, so they do not need initialization and data structures.

The taint variables list, taint import functions list, and dangerous functions list are built based on the Zend_Hash [32] API of the ZendVM. During the initialization, the taint import functions list and dangerous functions list add the user-configured name array of the taint import functions and dangerous functions to Zend_Hash, in order to improve the retrieval speed of the taint import functions and dangerous functions. The taint variables list in initialization module only finishes the initialization of the memory space, and does not insert any data. Variables in the Opline sequences are displayed in sequential Arabic numerals, and when a variable is marked as a taint variable, the Arabic number representing that variable is inserted into the taint variables list.

The taint propagation chain is constructed by a common doubly linked list. In fact, the taint propagation chain is an array storing the doubly linked list. Whenever a taint source is imported, a new doubly linked list will be created, and the head node of the linked list is the variable just imported by the taint source. When each taint variable (thisVar) is propagated to the next taint variable (nextVar), it will determine whether there is thisVar in the propagation chains according to the propagation relationship and insert nextVar into the next node of thisVar (Situation I). If it is found that thisVar node is not the tail node of the taint propagation chain, but the middle node (which means that the taint propagation chain is divided into two or more paths), then it will copy a new taint propagation list with thisVar as the tail node and insert nextVar into the next node of thisVar (Situation II). This is shown in Figure 3.

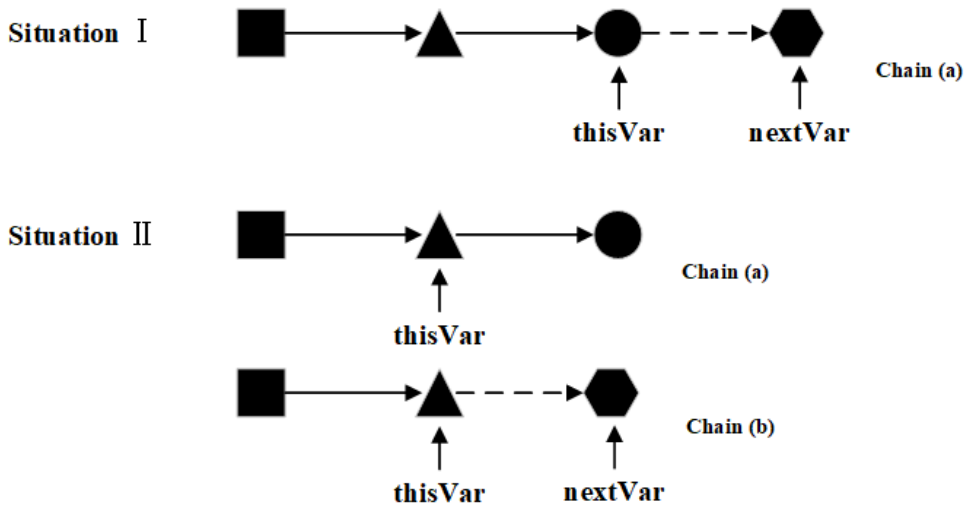

Figure 3. Taint propagation chain. Chain (a) consists of square triangles and circles, and the hexagon (nextVar) is the newly inserted node. In Situation II, chain (b) is generated from chain (a)'s header to thisVar, and then the new node (nextVar) is inserted into chain (b).

5.2. Taint Propagation Rules

The ZendVM has a unique instruction set of Oplines. Taint analysis based on Oplines needs a user-defined data flow logic. This section first introduces three definitions: Taint Attribute, Taint Map, and Predefined Taint. Next, it introduces the data flow logic of the ZendVM.

Definition 1 (Taint Attribute)**.** *Taint Attribute is an accessoryattribute of a variable in the Opline sequences and is a Boolean value. When a variable's Taint Attribute is True, it is a taint variable, and when its Taint Attribute is False, it is a normal variable.*

Definition 2 (Taint Map T(·))**.** *Regard v as a variable. T(v) will return the value of the variable v's Taint Attribute. The semantics of T(v) is related to the position of ←. When $T(v_1)$ is on the right of ←, it represents the acquisition of v_1's Taint Attribute. When $T(v_1)$ is on the left of ←, it represents the reception of the Taint Attribute, which represents that v_1's Taint Attribute is set to a Taint Attribute on the right. For example, $T(v_1) \leftarrow T(v_2)$ means v_2's Taint Attribute is passed to v_1.*

Definition 3 (Predefined Taint)**.** *Predefined Taint TAINT is a variable that has been pre-identified as a taint due to the characteristics of the PHP language and the ZendVM. Predefined Taints in this method specifically refer to super global variables [33] and the parameters of user-defined functions.*

Table 1 details the taint propagation logic. We deeply study the ZendVM instruction set, analyze the most probable taint propagation instructions, and finally, obtain this taint propagation logic. This table shows the taint import rules and taint propagation rules when our taint analysis system deals with the ZendVM instructions. The propagation rule of FETCH_R v_A, C is TAINT, because the Opline format only appears when super global variables are used. The propagation rule of RECV also has TAINT, because the ZendVM does not recompile PHP library functions, while RECV only occurs in function definition. Therefore, the occurrence of RECV means that the function is a user-defined function. Therefore, using the parameters of the user-defined function as TAINT is helpful for the user-defined function's taint analysis.

Table 1. Taint propagation logic.

Opline Format	Semantics	Taint Propagation	Description
ASSIGN v_A, C	$v_A \leftarrow C$	$T(v_A) \leftarrow \varnothing$	Clear v_A taint
ASSIGN v_A, v_B	$v_A \leftarrow v_B$	$T(v_A) \leftarrow T(v_B)$	Taint v_B propagate to v_A
ASSIGN_CONCAT v_A, v_B	$v_A \leftarrow Concat(v_A, v_B)$	$T(v_A) \leftarrow T(v_B)$	Taint v_B propagate to v_A
FAST_CONCAT v_A, v_B	$ret \leftarrow Concat(v_A, v_B)$	$T(ret) \leftarrow T(v_A) \cup T(v_B)$	Taint $(v_A \cup v_B)$ propagate to *ret*
CONCAT v_A, v_B	$ret \leftarrow Concat(v_A, v_B)$	$T(ret) \leftarrow T(v_A) \cup T(v_B)$	Taint $(v_A \cup v_B)$ propagate to *ret*
CAST v_A	$ret \leftarrow v_A$	$T(ret) \leftarrow T(v_A)$	Taint v_A propagate to *ret*
FETCH_R v_A, C	$v_A \leftarrow C$	$T(v_A) \leftarrow TAINT$	Taint superglobalsVar propagate to v_A
FETCH_R v_A, v_B	$v_A \leftarrow v_B$	$T(v_A) \leftarrow T(v_B)$	Taint v_B propagate to v_A
FETCH_DIM_R v_A, C	$ret \leftarrow v_A[C]$	$T(ret) \leftarrow T(v_A[C])$	Taint $v_A[C]$ propagate to *ret*
FETCH_DIM_R v_A, v_B	$ret \leftarrow v_A[v_B]$	$T(ret) \leftarrow T(v_A[v_B])$	Taint $v_A[v_B]$ propagate to *ret*
FETCH_OBJ_R v_A, C	$ret \leftarrow (v_A \rightarrow C)$	$T(ret) \leftarrow T(v_A \rightarrow C)$	Taint $v_A \rightarrow C$ propagate to *ret*
FETCH_OBJ_R v_A, v_B	$ret \leftarrow (v_A \rightarrow v_B)$	$T(ret) \leftarrow T(v_A \rightarrow v_B)$	Taint $v_A \rightarrow v_B$ propagate to *ret*
FETCH_FUNC_ARG v_A	$ret \leftarrow v_A$	$T(ret) \leftarrow T(v_A)$	Taint v_A propagate to *ret*
FETCH_DIM_FUNC_ARG v_A, C	$ret \leftarrow v_A[C]$	$T(ret) \leftarrow T(v_A[C])$	Taint $v_A[C]$ propagate to *ret*
FETCH_DIM_FUNC_ARG v_A, v_B	$ret \leftarrow v_A[v_B]$	$T(ret) \leftarrow T(v_A[v_B])$	Taint $v_A[v_B]$ propagate to *ret*
INIT_ARRAY v_A	$ret \leftarrow v_A$	$T(ret) \leftarrow T(v_A)$	Taint v_A propagate to *ret*
ADD_ARRAY_ELEMENT v_A	$ret \leftarrow v_A$	$T(ret) \leftarrow T(v_A)$	Taint v_A propagate to *ret*
RECV	$ret \leftarrow TAINT$	$T(ret) \leftarrow TAINT$	User-defined-func parameters
ASSIGN_DIM v_A, C *OP_DATA* v_C	$v_A[C] \leftarrow v_C$	$T(v_A[C]) \leftarrow T(v_C)$	Taint v_C propagate to $v_A[C]$
FETCH_DIM_W v_A, v_B *ASSIGN_DIM* v_C, v_D *OP_DATA* v_E	$tmp \leftarrow v_A[v_B]$ $v_C \leftarrow tmp$ $v_E \leftarrow v_C[v_D]$	$T(v_E) \leftarrow T(v_A[v_B][v_C])$	Taint $v_A[v_B][v_C]$ propagate to v_E
ROPE_INIT v_A *ROPE_ADD* tmp, v_B *ROPE_END* tmp, v_C	$tmp \leftarrow v_A$ $tmp \leftarrow v_B$ $ret \leftarrow Concat(tmp, v_C)$	$T(ret) \leftarrow T(v_A) \cup T(v_B) \cup T(v_C)$	Taint $(v_A \cup v_B \cup v_C)$ propagate to *ret*

5.3. Taint Sink Rules

The taint sink needs to meet two conditions: first, the call of dangerous function is detected; second, the called dangerous function uses the taint variable as the parameter. Similarly, it also needs a set of unique taint sink rules to judge the taint sink.

Table 2 provides a detailed list of taint sink rules. Oplines related to the taint sink mainly fall into three categories, namely function call initialization (INIT), passing parameters to the function (Param), and function call execution (CALL), which correspond to the three steps of function call execution in the ZendVM.

Table 2. Taint sink logic.

Opline	Type	SinkFuncFlag	TaintVarFlag	Sink
INIT_FCALL	INIT	•	–	–
INIT_FCALL_BY_NAME	INIT	•	–	–
INIT_METHOD_CALL	INIT	•	–	–
INIT_USER_CALL	INIT	•	–	–
INIT_DYNAMIC_CALL	INIT	•	–	–
SEND_VAL	Param	–	•	–
SEND_VAL_EX	Param	–	•	–
SEND_VAR	Param	–	•	–
SEND_VAR_EX	Param	–	•	–
SEND_USER	Param	–	•	–
SEND_ARRAY	Param	–	•	–
DO_ICALL	CALL	–	–	*SinkFuncFlag*&&*TaintVarFlag*
DO_FCALL	CALL	–	–	*SinkFuncFlag*&&*TaintVarFlag*
EVAL	CALL	–	•	*TaintVarFlag*

• means SinkFuncFlag or TaintVarFlag = 1, – means SinkFuncFlag or TaintVarFlag = 0.

In the phase of INIT, the corresponding operand of the Opline is detected. When the function called is found in the dangerous functions list, the value of sinkFuncFlag is set to one, indicating that the dangerous function is called.

In the phase of Param, when the corresponding operand of the Opline is found to be a taint variable, it is determined that the taint variable is imported by the function call, and the value of TaintVarFlag is set to one, indicating that the taint variable is passed as a parameter.

Finally, in the phase of CALL, when SinkFuncFlag = 1&&TaintVarFlag = 1 is found, it is the taint sink, and the sample is determined to be a webshell.

It is worth noting that Opline Eval itself represents a dangerous function call, so it only needs to meet TaintVarFlag = 1 to qualify as a webshell. In addition, SinkFuncFlag and TaintVarFlag always appear in pairs, and both Flag values are reset to zero after the judgment is completed.

5.4. Example Illustration

Reviewing the include-type webshell example named webshell-1.php in Section 3 and analyzing it using the method proposed in this paper, the sample code is transformed into Opline sequences, as shown in Figure 4. In this example, hello.txt is the file included in the webshell-1.php. The Opline sequences of hello.txt are obtained by file inclusion recursion detection, which is shown in Figure 5.

```
#* E I O op                          fetch          ext  return  operands
---------------------------------------------------------------------------
0  E >   INCLUDE_OR_EVAL                                          'hello.txt', INCLUDE
1      > RETURN                                                   1
```

Figure 4. The Opline sequences of webshell1.php. Columns 5 (op) and 8 (return) represent opcode and opline return values. Column 9 (operands) has three values, oprands1, oprands2, and extended_value.

```
#* E I O op                          fetch          ext  return  operands
---------------------------------------------------------------------------
0  E >   INIT_FCALL                                               'md5'
1        FETCH_R                     global             $0      '_GET'
2        FETCH_DIM_R                                    $1      $0, 'pass'
3        SEND_VAR                                               $1
4        DO_ICALL                                       $2
5        IS_EQUAL                                       ~3      $2,'21232f297a57a5a743894a0e4a801fc3'
6      > JMPZ                                                   ~3, ->12
7    >   FETCH_CONSTANT                                 ~5      'console'
8        FETCH_R                     global             $4      '_POST'
9        FETCH_DIM_R                                    $6      $4, ~5
10       INCLUDE_OR_EVAL                                        $6, EVAL
11     > JMP                                                    ->13
12   > > EXIT                                                   'exit'
13   > > RETURN                                                 1
```

Figure 5. The Opline sequences of hello.txt.

In Figure 5, it is observed that Line 8 is in accordance with the taint propagation rule of FETCH_R v_A, C, and C is the super global variable _POST, so Variable 4 is added to the taint variable list and a taint propagation chain is created at the same time, with Variable 4 as the head node of this chain. Then, the Opline in Line 9 conforms to the taint propagation rule of FETCH_DIM_R v_A, C, so the return value of Variable 6 is tainted, added to the taint variable list, and inserted into the taint propagation chain with Variable 4 as the previous node. On Line 10, the extended_value of Opline is EVAL, and op1 is Variable 6, which is a taint variable. Therefore, it meets the taint sink rules, and the sample file is determined to be a webshell file.

The method proposed in this paper can be used to easily determine that the sample code is a webshell. We will further verify the advantages of our method in the next section.

6. Evaluation

In order to evaluate the webshell static detection method based on taint analysis proposed in this paper, a series of experiments is designed on the real program in this section and compared with relevant technologies. The experiments are described below.

6.1. Evaluation Setup

We designed the experiments to answer the following research questions:

RQ1: Is the interprocedural analysis module based on taint analysis effective at the detection of user-defined function-type webshells and include-type webshells?

RQ2: Does the webshell detection method based on taint analysis have a better effect against unknown webshells?

RQ3: Does WTA have better performance than well-known webshell detection tools?

The first two experiments were used to evaluate the two improved techniques proposed in this paper, and the third experiment was used to evaluate the overall performance of the method proposed in this paper.

Experimental infrastructure. All experiments were run on a machine with an Intel Core i7-10875h processor, four 2.30 GHz logic cores, and 16 GB of RAM, and the operating system was 64-bit Windows10 20H2 or 64-bit Linux Ubuntu 18.04. The PHP version was 7.1.24.

6.2. Evaluation Benchmarks

There are many publicly available webshell datasets on the Internet that can be obtained through GitHub. Since the collection of these datasets is random, there are problems such as sample duplication, sample execution failure, and incorrect sample format. In addition, the research object of this paper is PHP webshells. After cleaning, we collected a total of 1776 PHP webshells from 10 open-source datasets on GitHub. The sources of the samples are shown in Table 3.

Table 3. The sources of the webshells (Accessed on 7 January 2021).

No	Webshell Projects	Source
1	tennc/webshell	https://github.com/tennc/webshell
2	JohnTroony/php-webshells	https://github.com/JohnTroony/php-webshells
3	ysrc/webshell-sample	https://github.com/ysrc/webshell-sample
4	xl7dev/WebShell	https://github.com/xl7dev/WebShell
5	BlackArch/webshells	https://github.com/BlackArch/webshells
6	LandGrey/webshell-detect-bypass	https://github.com/LandGrey/webshell-detect-bypass
7	backlion/webshell	https://github.com/backlion/webshell
8	x-o-r-r-o/PHP-Webshells-Collection	https://github.com/x-o-r-r-o/PHP-Webshells-Collection
9	tdifg/WebShell	https://github.com/tdifg/WebShell
10	s0md3v/nano	https://github.com/s0md3v/nano

These datasets from GitHub have different purposes for collecting webshells. Some of them aim to collect the most comprehensive webshells, so in addition to the language PHP, the languages of the samples also include ASP, Java, Python, and so on, such as tennc/webshell. Some divide PHP webshells according to their families and only collect webshells with typical family characteristics, such as S0MD3v/Nano. Some, such as LandGrey/webshell-Detect-Bypass, collect webshells that can bypass current detection methods for the purpose of network attack or security research. Therefore, we preprocessed the collected samples for the above 10 projects: repeated samples were excluded based on the SHA1 algorithm. In addition, PHPCLI was used to execute each PHP webshell and excluded some webshells that cannot be executed. Finally, we obtained 1776 executable PHP webshell samples.

Using the same method, we collected 6874 PHP web pages from six popular open-source PHP Content Management Systems (CMSs), and Table 4 shows the CMS information.

Table 4. The sources of the CMSs (Accessed on 13 May 2021).

No	CMS	Source
1	joomla	https://github.com/joomla/joomla-cms/releases/download/3.9.26/Joomla_3.9.26-Stable-Full_Package.zip
2	laravel	https://github.com/laravel/laravel/archive/refs/tags/v8.5.17.zip
3	phpBB	https://github.com/phpbb/phpbb/archive/refs/tags/release-3.3.4.zip
4	typecho	https://github.com/typecho/typecho/archive/refs/tags/v1.1-17.10.30-release.zip
5	thinkphp	http://www.thinkphp.cn/download/1278.html
6	october	https://github.com/octobercms/october/archive/refs/tags/v2.0.14.zip

Besides, we selected eight currently popular webshell detection systems for the controlled experiment with WTA. They were D-Shield [15], ShellPub [34], WebshellKiller [35] (precision mode), WebshellKiller [35] (recall mode), CloudWalker [36], ClamAV [37], LoKi [38], and findbot.pl [39]. The basic information and basic detection principles of these tools are shown in Table 5.

Table 5. Webshell detection tools.

Webshell Detection Tool	Version	Description
D-shield	V2.1.5.4	A webshell detection tool based on feature detection.
SHELLPUB	V1.8.2	Antivirus technology using traditional features and dual-engine cloud big data.
WebshellKiller (precision/recall mode)	V3.3.0.2	A webshell detection tool adopting simulation execution, parameter dynamic analysis and monitoring technology, and webshell semantic analysis technology. Precision mode and recall mode are the two performance modes of the tool. Different parameter thresholds are set to improve precision and recall, respectively.
CloudWalker	V1.0.0	A comprehensive webshell detection tool combined with feature detection and machine learning.
ClamAV	V0.103.3	ClamAV is an open-source antivirus engine for detecting Trojans, viruses, malware, and other malicious threats.
LoKi	V0.41.0	LOKI is a free and simple IOC and YARA scanner.
Findbot.pl	V0.10	A script attempts to find malicious files/scripts on your machine.

6.3. Effectiveness Test of the Interprocedural Analysis Module Based on Taint Analysis (RQ1)

To evaluate the effectiveness of the interprocedural analysis module, we implemented two versions of the webshell static detection tool: Webshell Taint Analysis (WTA) and No Interprocedural Analysis Module (WTA-NO-IAM). The former uses the method proposed in this paper, while the latter does not include the interprocedural analysis module. In this experiment, there were two performance evaluation indicators. First, validating sample code was set up to test the validity of two types, user-defined function-type webshell and include-type webshell. Second, WTA and WTA-NO-IAM were applied to detect our webshell dataset, and the effectiveness of the interprocedural analysis module was evaluated by comparison of the number of webshells detected by the two tools.

The validating sample code for the webshell of the user-defined function-type webshell (WebShell-1) and the include-type webshell (WebShell-2) is shown in Listing 5.

As shown in Listing (a), the sample webshell encapsulates the easy-to-detect keyword "_POST" in the user-defined function "dynamic" and the easy-to-detect dangerous function "assert" in the user-defined function newassert(), trying to bypass the detection of the webshell detection tool. Finally, at Line 16, they are concatenated into a one-sentence webshell: assert($_POST[' x ']).

The include-type webshell is shown in Lists (b) and (c), where List (b) is the body of the webshell and List (c) is the content of the included file. The sample webshell uses "include" to wrap the dangerous function "eval" into the user-defined function "HelloWorld" and calls "HelloWorld" in the body of the webshell file. The webshell is finally achieved at Line 3 of the list (b): eval ($_POST['hello']).

WTA, WTA-NO-IAM, and famous webshell tools were used to detect the two webshells, and the test results are shown in Table 6.

Listing 5. The validating sample code.

```
1.<?php
2.function dynamic(){
3.        $a = "_POST";
4.        return $a;
5.}
6.
7.function newassert(){
8.        $e = "a###sse###rt";
9.        $f = chunk_split($e,1,"#");
10.       $g = str_replace("#","",$f);
11.       return $g;
12.}
13.$a = dynamic();
14.$b = $$a;
15.$c = newassert();
16.$c ($b['x']);
17.?>

(a) User-defined function-type (Webshell-1)


1.<?php
2.include("../eval.php");
3.helloworld($_POST['hello']);
4.?>

(b) Include-type (Webshell-2)


1.<?php
2.function helloworld($a){
3.        eval($a);
4.}
5.?>

(c) eval.php included
```

For the detection of the sample webshells, it can be observed from the experimental results that both samples could be detected by WTA, indicating that WTA can detect the user-defined function-type webshells and include-type webshells. For the other tools, only D-Shield could detect webshell-1 and report it as suspicious at Level 1 (D-Shield detects webshells on a scale of five, with Level 1 being the least dangerous). D-Shield judged it as a Level-1 webshell because it detected variable function [40] $\$c(\$b['x'])$. This shows that the performance of the regular matching detection method adopted by the current detection tools is weak at interprocedural analysis, especially for include-type webshells.

Table 6. Interprocedural analysis of the test code experiment effect.

Webshell Detection Tool	Webshell-1	Webshell-2
WTA (Webshell Taint Analysis)	✓	✓
WTA-NO-IAM (No Interprocedural Analysis Module)	×	×
D-Shield	✓(level 1)	×
SHELLPUB	×	×
WebshellKiller	×	×
WebshellKiller	×	×
CloudWalker	×	×

WTA and WTA-NO-IAM are used to detect the webshell dataset, and the test results are shown in Table 7.

Table 7. Performance comparison between WTA and WTA-NO-IAM.

Webshell Detection Tool	TP	FN	Recall
WTA(Webshell Taint Analysis)	1713	63	0.964527027
WTA-NO-IAM(no interprocedural analysis module)	1325	451	0.746058558

For the controlled experiment of WTA and WTA-NO-IAM, the recall rate of WTA was 96.4%, while the recall rate of WTA-NO-IAM was only 74.6%. Therefore, the interprocedural analysis module of WTA plays a crucial role in the detection of webshell samples.

To sum up, the answer to RQ1 is obvious. The interprocedural analysis module based on taint analysis is effective at the detection of user-defined function-type webshells and include-type webshells.

6.4. The Validity Test of the Webshell Detection Method Based on Taint Analysis against Unknown Webshells (RQ2)

At present, the mainstream webshell detection tools mostly use the detection method based on regular matching. By capturing the webshells in the wild, the corresponding features are extracted and added into the feature library. Therefore, the webshells that have been spread on the network for a period of time are easier to detect. Moreover, this method of blacklist matching is easy to bypass. How to improve the detection ability against unknown webshells is a goal pursued by various webshell detection tools.

To evaluate the effectiveness of this detection method against unknown webshells, we found 5 generation tools that can generate antidetect PHP webshells randomly and used each tool to generate 10 webshells, respectively. Therefore, there was a total of 50 samples. The information on the 5 webshell generation tools is shown in Table 8.

Table 8. The sources of the antidetect webshell generation tools.

Tool	Source	Last Update
pureqh	https://github.com/pureqh/webshell	12 February 2021
venom	https://github.com/0x6b7966/webshell-venom	2 July 2019
weevely	https://github.com/epinna/weevely3	8 August 2020
b374k	https://github.com/b374k/b374k	13 December 2016
aqk	https://www.anquanke.com/post/id/193042	21 November 2019

Listing 6 shows the code snippet of pureqh. There are many anchors for replacement in the code, such as {1}, {6}, {7}. These anchor will be replaced with random strings when pureqh runs. Therefore, the webshell was generated each time with different hash values. It is difficult for current static detection tools to extract the features, resulting in detection failure.

The popular webshell detection tools mentioned in Table 8 were used to detect the 50 samples, and the detection results are shown in Table 9.

From the experimental results, it can be observed that WTA had an excellent detection effect for the randomly generated unknown webshells. All 50 samples could be detected, and the recall rate reached 100%. Since WTA adopts the taint analysis method for detection, there was no need to extract the corresponding features, and the detection effect was better for unknown webshells. Among the well-known webshell detection tools, only D-Shield, WebshellKiller (recall mode), and CloudWalker could find the webshells. D-Shield could detect 40 webshells, but could not detect webshells generated by pureqh. WebshellKiller (recall mode) could only detect the samples generated by weevely and b374k; other samples could not be detected. CloudWalker could only find 10 webshells.

Listing 6. The validating sample code.

```
1.function {6}(${7}){1}
2.      $BASE32_ALPHABET = 'abcdefghijklmnopqrstuvwxyz234567';
3.      ${8} = '';
4.      $v = 0;
5.      $vbits = 0;
6.      for ($i = 0, $j = strlen(${7}); $i < $j; $i++){1}
7.      $v <<= 8;
8.      $v += ord(${7}[$i]);
9.      $vbits += 8;
10.     while ($vbits >= 5) {1}
11.     $vbits -= 5;
12.     ${8}.= $BASE32_ALPHABET[$v >> $vbits];
13.     $v &= ((1 << $vbits) - 1);{4}{4}
14.     if ($vbits > 0){1}
15.     $v <<= (5 - $vbits);
16.     ${8}.= $BASE32_ALPHABET[$v];{4}
17.     return ${8};{4}$
```

Table 9. The detection effect for unknown webshells.

	The Number of Webshells	D-Shield	SHELLPUB	WebshellKiller (Precision)	WebshellKiller (Recall)	CloudWalker	WTA
venom	10	10	0	0	0	0	10
weevely	10	10	0	0	10	0	10
b374k	10	10	0	0	10	10	10
aqk	10	10	0	0	0	0	10
ureqh	10	0	0	0	0	0	10
total	10	40	0	0	20	10	50
Recall	–	80%	0%	0%	40%	20%	100%

It is worth mentioning that D-Shield's detection report stated that the 40 samples were "known webshells", indicating that D-Shield only noticed the four generation tools and added their features to their webshell feature library. The updated time of these tools is also a good indication of this viewpoint. The detected webshells were all generated by tools updated before 2021. The oldest tool, b374k, was last updated on 13 December 2016. As for the webshells generated by the latest tool pureqh, D-Shield could not detect them, while our WTA based on taint analysis could achieve a better detection effect for unknown webshells.

To sum up, the answer to RQ2 is evident. The webshell detection method based on taint analysis has a better effect against unknown webshells and can provide important help for detecting webshells.

6.5. WTA and Well-Known Webshell Detection Tools for Performance Comparison (RQ3)

The above two experiments evaluated the effectiveness of the two key techniques in this paper. The experiment in this section evaluated the overall performance of the system and whether the method presented in this paper can improve the performance of webshell detection.

In order to better evaluate the performance of webshell detection tools, the evaluation indicators of this experiment are defined as follows:

We regarded webshells as positive samples and normal files as negative samples;

True Positive (TP). The webshell sample is correctly recognized as a webshell;

False positive (FP). The normal file is misidentified as a webshell;

True Negative (TN). The normal file is correctly recognized as a normal file;

False Negative (FN). The webshell sample is misidentified as a normal file;

Accuracy. The proportion of correctly predicted samples to all samples. The formula is as follows:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}}; \quad (1)$$

Recall. The proportion of correctly predicted webshell samples to the real webshell samples; the higher the recall rate, the better the performance is for potential webshells' detection. The formula is as follows:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}; \tag{2}$$

Precision. The proportion of correctly predicted webshell samples to the predicted webshell samples; the higher the precision rate, the lower the false positive rate is. The formula is as follows:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}; \tag{3}$$

F-measure. The F-measure is a comprehensive consideration of recall and precision. Generally, a higher F1 indicates that the experimental method is more effective. The larger β is, the more importance is attached to the recall. The β values used in our experiment were 0.5, 1, and 1.5. The formula is as follows:

$$F_\beta = (1 + \beta^2)\frac{\text{Precision} * \text{Recall}}{(\beta^2 * \text{Precision}) + \text{Recall}}. \tag{4}$$

In the experiment, the method presented in this paper was compared with well-known webshell detection tools, and a controlled experiment was conducted based on the experimental dataset in Section 6.2. The experimental results are shown in Table 10.

Table 10. Comparison of the webshell detection tools.

	WTA	**D-Shield**	**SHELLPUB**	**WebshellKiller (P)**	**WebshellKiller (R)**	**CloudWalker**	**LoKi**	**ClamAV**	**findbot.pl**
TP	1713	1608	1165	897	1376	1580	1474	391	972
FN	63	168	611	879	400	196	302	1385	804
FP	40	3	75	2	53	30	10	1	132
TN	6834	6870	6798	6871	6820	6843	6863	6872	6741
Accuracy	**0.9881**	0.9802	0.9207	0.8981	0.9476	0.9739	0.9639	0.8398	0.8918
Recall	**0.9645**	0.9054	0.6560	0.5051	0.7748	0.8896	0.8300	0.2206	0.5473
Precision	**0.9772**	0.9981	0.9395	0.9978	0.9629	0.9814	0.9933	0.9974	0.8804
$F_{0.5}$	**0.9746**	0.9781	0.8648	0.8349	0.9183	0.9615	0.9557	0.5846	0.7849
F_1	**0.9708**	0.9495	0.7725	0.6707	0.8587	0.9332	0.9043	0.3607	0.6750
$F_{1.5}$	**0.9684**	0.9320	0.7231	0.5956	0.8243	0.9159	0.8742	0.2896	0.6194

The experimental results showed that D-Shield had the best comprehensive performance among the well-known webshell detection tools, whose recall was 90.54%, precision 99.81%, and F_1 94.95%. However, our system WTA had better comprehensive performance than D-Shield, with a recall of 96.45%, which was 5.91% higher than D-Shield, precision 97.71%, slightly lower than D-Shield, but also a high level of precision, and F_1 97.08%, which was 2.13% higher than D-Shield. Obviously, the performance of our method was at the top of all webshell detection tools.

It is worth noting that the performance of the two modes in WebshellKiller was quite different, with a precision of 99.77% in precision mode. While some precision was sacrificed in recall mode, the recall was 26.97% higher than precision mode, but still only 77.47%, which is an average performance. SHELLPUB's detection speed was the fastest among all the tools: the detection of webshell samples took less than 10 seconds; however, its recall was too low. CloudWalker applies a number of detection techniques, such as statistical feature detection, AST detection, regular matching, machine learning, etc. Therefore, its detection speed was the slowest, and the average detection time was three-times that of other tools.

6.6. Discussion

In this section, we discuss the limitations and future developments of static webshell detection methods based on taint analysis to improve the integrity of WTA.

Although our webshell static detection method based on taint analysis analysis had a more complete interprocedural capability than traditional methods and could detect more unknown webshells, it still could not guarantee that it could detect all new unknown webshells. Many factors affect WTA's detection of unknown webshells, such as the propagation rules of the Opline in this paper not being totally comprehensive, some webshells using new PHP features, and so on. In future work, we will further study the taint propagation rules of the Opline, which are not involved at present, and expand the current taint propagation logic. In addition, we will update the taint propagation logic of the Opline in the new version of PHP by updating to it and continue to study the principles and features of new webshells caused by the features of the new version of PHP.

In addition, our method is extensible. Specifically, we will further expand the detection objects, such as web application vulnerabilities, SQL injection vulnerabilities, XSS vulnerabilities, and so on. For example, we can build the taint import rules of SQL injection, improve the corresponding taint propagation logic, and find the dangerous functions list of SQL injection separately, then finally realize the detection of SQL injection vulnerabilities. In this paper, the effectiveness of our method in webshell detection was evaluated in a preliminarily fashion. In the future, we will expand to web application vulnerability detection, such as SQL injection vulnerability detection, XSS vulnerability detection, and so on.

7. Related Works

In this section, some dynamic feature-based and static feature-based detection methods are introduced, respectively.

7.1. Dynamic Methods

Tian et al. [9] proposed a malicious webshell detection method based on a Convolutional Neural Network (CNN). This method first obtains the HTTP request and then uses word2vec to represent each word as a vector. In this case, each HTTP request can be transformed into a fixed-size matrix; finally, a model is trained to detect and classify a file based on CNN. In fact, this detection method is based on network traffic, which uses a convolutional neural network to monitor, model, and train the traffic at webshell runtime. This method has better classification performance than the method based on malicious keyword matching, but it also has some drawbacks: First, if attackers reduce the frequency of communication, such as disguising the operations as normal behaviors and executing the required command only once, it would easily bypass the detection based on network traffic. Second, running the webshell in real time results in bad performance and consumes many computing resources, which may lead to the destruction of key nodes in the system.

In addition, there are methods to detect abnormal behaviors of the webserver to detect web attacks. The main detection idea is to extract the characteristics of abnormal network behaviors, distinguish them from the normal network behavior, and construct the abnormal network activity label for web attack detection. However, systems based on anomaly detection often produce a large number of false positives, because it is difficult to construct the algorithm for labeling normal and abnormal behaviors, and it is easy to mark normal behaviors as suspicious operations or omit some real abnormal behaviors. Robertson et al. [41] proposed a network attack detection method. This method uses exception generalization technology to convert suspicious web requests into abnormal signatures and then uses these signatures to group similar abnormal samples. Kruegel et al. [42] proposed an intrusion detection system that uses a variety of different anomaly detection techniques to detect attacks against web servers and web applications. Almgren et al. [43] took into account the characteristics of different types of host-based attacks and developed a lightweight tool for online detection of webserver attacks that can run and track suspected hosts in real-time.

7.2. Static Methods

Tian et al. [9] and Tu et al. [3] used regular matching and keyword feature matching to detect webshells. This method can be effective at identifying some webshells, but webshells are usually written in high-level languages, which have abstract lexical and syntactic features. These features cannot be fully reflected in regular expressions, so it is difficult to extract abstract features in this method, and there may be missing problems in the detection process.

Zhu et al. [44] considered the abstract lexical and syntactic features in high-level languages (especially the PHP language) and proposed a detection method based on multiview feature fusion. First, this method extracts the abstract features of the vocabulary and syntax that represent the internal meaning of the webshell. Secondly, the Fisher score is used to rank each feature according to its importance. Finally, a model is established based on the optimized Support Vector Machine (SVM), and it could detect webshells effectively.

The text feature recognition method is also the main method in webshell static detection, which often plays a role together with a neural network and deep learning. Tu et al. [45] proposed a webshell detection system based on a scoring mechanism, which determines whether suspicious files belong to a webshell by scoring. The factors of scoring are the function type, the number of dangerous functions, the signature status, the longest string length, and so on. Thresholds are then determined, and score accumulation is performed when some factors exceed the threshold. This method mainly has the following problems: first, because this method is mainly based on the feature library constructed by experts to determine dangerous functions and other factors, the new webshell cannot be detected; second, if the attacker encrypts or splits dangerous functions and sensitive parameters, it cannot be detected directly.

Each programmer's programming style results in different code syntax, and these syntactic variations are difficult points in taint analysis. Kurniawan et al. [46] summarized the possible syntax variants based on AST and reconstructed the PHP parser, which can reduce the syntax objects to be visited in the process of taint analysis. In contrast, our taint analysis method performs analysis on the PHP Opline. The Opline is a ZendVM instruction, so our method performs taint analysis on the PHP Opline, which naturally can resolve grammatical variants.

Le et al. [47,48] combined taint analysis and pattern matching to detect webshells. Taint analysis is performed to divide the code into tokens during the lexical analysis phase. Taint analysis is performed based on tokens, similar to RIPS [19], and pattern matching can match a few one-sentence webshells.

The method of statistical characteristics summarizes the characteristics of an entire webshell file according to the attribute values of certain aspects of the file. Due to the rapid development of web services, developers tend to use encryption and obtrusion techniques to avoid source code leakage, which leads to the statistical characteristics of normal files being similar to that of the webshell files. Therefore, a webshell detection method based on statistical characteristics loses its original advantages. Pan et al. [14] proposed a webshell detection method based on executable data features in PHP code. This method combines the characteristics of executable data from the PHP code with the characteristics of the static text to detect webshells. Compared to the traditional static statistical method, this method can improve the recognition ability.

Webshell detection systems will use different classification methods to determine whether a suspicious file is a webshell. For example, the webshell detection method proposed by Wang et al. [49] uses a multilayer neural network to detect and classify suspicious files. Cui et al. [50] used the combination of a random forest classifier and a GBDT classifier for classification. Fang et al. [21] used the fastText algorithm to train the Opcode sequence model and predicted the corresponding features of the samples. Finally, random forest was used to realize the binary classification. Each of these methods has its advantages and disadvantages. Ai et al. [51] proposed a webshell detection method based on ensemble learning, which constructed a differentiated ensemble detection model, WS-

LSMR, composed of Logistic Regression (LR), Support Vector Machine (SVM), Multilayer Perceptron (MLP), and Random Forest (RF). Given the four basic classifiers (LR, SVC, MLP, RF), this model adaptively assigns weights to the four classifiers, and algorithms with high accuracy will have high weights to better reflect the effect of good algorithms.

8. Conclusions

Webshells are an important threat to network security. Attackers using a webshell can invade websites, control servers, steal sensitive files, and further invade the internal network. How to improve the capability of interprocedural analysis and improve the detection ability for unknown webshells are the main challenges of webshell detection. This paper proposed a webshell static detection method based on taint analysis. For the first time, we constructed a set of user-defined taint propagation rules and a set of user-defined taint sink rules for the unique instruction set of the ZendVM. A PHP webshell detection method was formed by the combination of the two sets of rules and the detection of the Opline taint source. Based on this method, we implemented a static taint analysis prototype system named WTA for the detection of PHP webshells.

Experimental results showed that WTA supports interprocedural analysis and has the ability to detect unknown webshells. Compared with the current popular webshell detection tools, WTA can detect more webshells. Its recall rate reached 96.45%, which was 5.91% higher than the best-performing tool among the other tools. The precision rate was 97.71%, and the F_1 was 97.08%.

Author Contributions: Conceptualization, Y.L. and J.Z.; methodology, J.Z., X.W. and K.Z.; software, J.Z. and L.Y.; validation, J.Z. and X.W.; investigation, J.Z.; resources, Y.L.; data curation, J.Z. and X.W.; writing—original draft preparation, J.Z.; writing—review and editing, Y.L., K.Z., L.Y. and J.Z.; supervision, Y.L.; project administration, Y.L. All authors read and agreed to the published version of the manuscript.

Funding: This research was supported by the National Key Research and Development Project of China (No. 2017YFB0802900).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Acknowledgments: We would like to sincerely thank the reviewers for their insightful comments, which helped us improve this work.

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

References

1. Al-Fedaghi, S. Developing web applications. *Int. J. Softw. Eng. Its Appl.* **2011**, *5*, 57–68.
2. microsoft. Microsoft Defender Advanced Threat Protection. Available online: https://docs.microsoft.com/en-us/windows/security/threat-protection/microsoft-defender-atp/microsoft-defender-advanced-threat-protection (accessed on 20 March 2021).
3. Tu, T.D.; Cheng, G.; Guo, X.; Pan, W. Webshell detection techniques in web applications. In Proceedings of the Fifth International Conference on Computing, Communications and Networking Technologies (ICCCNT), Hefei, China, 11–13 July 2014; pp. 1–7.
4. Xie, Y.; Aiken, A. Static Detection of Security Vulnerabilities in Scripting Languages. In Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, 31 July–4 August 2006; Volume 15, pp. 179–192.
5. Sommestad, T.; Holm, H.; Ekstedt, M. Estimates of success rates of remote arbitrary code execution attacks. *Inf. Manag. Comput. Secur.* **2012**, *20*, 107–122. [CrossRef]
6. Hannousse, A.; Yahiouche, S. Handling webshell attacks: A systematic mapping and survey. *Comput. Secur.* **2021**, *108*, 102366. [CrossRef]
7. Sun, X.; Lu, X.; Dai, H. A matrix decomposition based webshell detection method. In Proceedings of the 2017 International Conference on Cryptography, Security and Privacy, Wuhan, China, 17–19 March 2017; pp. 66–70
8. Wei, Y.B.; Huang, J.Q.; Zhou, X. PHP Technology and It's Application. *Comput. Mod.* **2000**, *5*, 86–89.
9. Tian, Y.; Wang, J.; Zhou, Z.; Zhou, S. CNN-webshell: Malicious web shell detection with convolutional neural network. In Proceedings of the 2017 VI International Conference on Network, Communication and Computing, Kunming, China, 8–10 December 2017; pp. 75–79

10. Yong, B.; Liu, X.; Liu, Y.; Yin, H.; Huang, L.; Zhou, Q. 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), Guangzhou, China, 8–12 October 2018; pp. 1911–1916.
11. Liuyang, S.; Yong, F. Webshell detection method research based on web log. *J. Netw. New Media* **2016**, *2*, 66–73.
12. Wu, Y.; Sun, Y.; Huang, C.; Jia, P.; Liu, L. Session-based webshell detection using machine learning in web logs. *Secur. Commun. Netw.* **2019**, *2019*, 3093809. [CrossRef]
13. Thompson, K. Programming techniques: Regular expression search algorithm. *Commun. ACM* **1968**, *11*, 419–422. [CrossRef]
14. Pan, Z.; Chen, Y.; Chen, Y.; Shen, Y.; Guo, X. Webshell Detection Based on Executable Data Characteristics of PHP Code. *Wirel. Commun. Mob. Comput.* **2021**, *2021*, 5533963. [CrossRef]
15. D-Shield. D-Shield. Available online: http://www.d99net.net/ (accessed on 14 February 2021).
16. Guo, Y.; Marco-Gisbert, H.; Keir, P. Mitigating webshell attacks through machine learning techniques. *Future Internet* **2020**, *12*, 12. [CrossRef]
17. Newsome, J.; Song, D.X. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. In Proceedings of the NDSS Symposium 2005, San Diego, CA, USA, 3 February 2005; Volume 5, pp. 3–4.
18. Yang, Z.; Yang, M. Leakminer: Detect information leakage on android with static taint analysis. In Proceedings of the 2012 Third World Congress on Software Engineering, Wuhan, China, 6–8 November 2012; pp. 101–104.
19. Dahse, J.; Holz, T. Simulation of Built-in PHP Features for Precise Static Code Analysis In Proceedings of the NDSS Symposium 2014, San Diego, CA, USA, 23–26 February 2014; Volume 14, pp. 101–104.
20. Li, Y.; Huang, J.; Ikusan, A.; Mitchell, M.; Zhang, J.; Dai, R. ShellBreaker: Automatically detecting PHP-based malicious web shells. *Comput. Secur.* **2019**, *87*, 101595. [CrossRef]
21. Fang, Y.; Qiu, Y.; Liu, L.; Huang, C. Detecting webshell based on random forest with fasttext. In Proceedings of the 2018 International Conference on Computing and Artificial Intelligence, Chengdu, China, 12–14 March 2018; pp. 52–56.
22. PHP. PHP: Hypertext Preprocessor. Available online: https://www.php.net/ (accessed on 20 April 2021).
23. Zend. Zend Engine. Available online: https://www.zend.com/ (accessed on 20 April 2021).
24. nikic. PHP 7 Virtual Machine. Available online: https://www.npopov.com/2017/04/14/PHP-7-Virtual-machine.html (accessed on 14 January 2021).
25. Zend. How to Use PHP Extensions. Available online: https://www.zend.com/blog/php-development-using-php-extensions (accessed on 14 February 2021).
26. Derickr. Vulcan Logic Dumper. Available online: https://derickrethans.nl/projects.html#vld (accessed on 14 January 2021).
27. Reps, T. Solving demand versions of interprocedural analysis problems. In *Compiler Construction*; Springer: Berlin/Heidelberg, Germany, 1994; pp. 389–403.
28. Zhao, Z.; Liu, Q.; Song, T.; Wang, Z.; Wu, X. WSLD: Detecting Unknown Webshell Using Fuzzy Matching and Deep Learning. In Proceedings of the International Conference on Information and Communications Security, Edinburgh, UK, 7–9 April 2019; Springer: Cham, Switzerland, 2019; pp. 725–745.
29. PHP. Function Reference. Available online: https://www.php.net/manual/en/funcref.php (accessed on 12 January 2021).
30. Day, J.; Matta, I.; Mattar, K. Networking is IPC: A guiding principle to a better internet. In Proceedings of the 2008 ACM CoNEXT Conference, Madrid, Spain, 9–12 December 2008; pp. 1–6.
31. Enck, W.; Gilbert, P.; Han, S.; Tendulkar, V.; Chun, B.G.; Cox, L.P.; Jung, J.; McDaniel, P.; Sheth, A.N. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst. (TOCS)* **2014**, *32*, 1–29. [CrossRef]
32. Php Internals Book. PHP Internals Book HASHTABLE API. Available online: https://www.phpinternalsbook.com/php5/hashtables/hashtable_api.html (accessed on 12 January 2021).
33. PHP. PHP Predefined Variables. Available online: https://www.php.net/manual/en/language.variables.superglobals.php (accessed on 12 January 2021).
34. Shellpub. Shellpub. Available online: https://www.shellpub.com/ (accessed on 14 February 2021).
35. sangfor. WebShellkiller. Available online: http://edr.sangfor.com.cn/tool/WebShellKillerTool.zip (accessed on 14 February 2019).
36. chaitin. Cloudwalker. Available online: https://github.com/chaitin/cloudwalker/releases/download/webshell-detector-1.0.0/webshell-detector-1.0.0-linux-amd64 (accessed on 14 February 2020).
37. Cisco-Talos. clamav. Available online: https://github.com/Cisco-Talos/clamav/archive/refs/tags/clamav-0.103.3.tar.gz (accessed on 13 July 2021).
38. Neo23x0. loki. Available online: https://github.com/Neo23x0/LoKi/releases/download/0.41.0/loki_0.41.0.zip (accessed on 12 July 2021).
39. CBL. findbot.pl. Available online: https://www.abuseat.org/findbot.pl (accessed on 12 July 2021).
40. PHP Variable Functions. PHP. Available online: https://www.php.net/manual/en/functions.variable-functions.php (accessed on 12 January 2021).
41. Robertson, W.; Vigna, G.; Kruegel, C.; Kemmerer, R.A. Using Generalization and Characterization Techniques in the Anomaly-Based Detection of Web Attacks. In Proceedings of the NDSS Symposium 2006, San Diego, CA, USA, 2 February 2006.

42. Kruegel, C.; Vigna, G. Anomaly detection of web-based attacks. In Proceedings of the 10th ACM Conference on Computer and Communications Security, Washington, DC, USA, 27–30 October 2003; pp. 251–261.
43. Almgren, M.; Debar, H.; Dacier, M. A Lightweight Tool for Detecting Web Server Attacks. In Proceedings of the NDSS Symposium 2000, San Diego, CA, USA, 3–4 February 2000.
44. Zhu, T.; Weng, Z.; Fu, L.; Ruan, L. A Web Shell Detection Method Based on Multiview Feature Fusion. *Appl. Sci.* **2020**, *10*, 6274. [CrossRef]
45. Tu, T.D.; Cheng, G.; Guo, X.; Pan, W. Evil-hunter: A novel web shell detection systembased on scoring scheme. *J. Southeast Univ. (Engl. Ed.)* **2014**, *30*, 278–284.
46. Kurniawan, A.; Abbas, B.S.; Trisetyarso, A.; Isa, S.M. Static Taint Analysis Traversal with Object Oriented Component for Web File Injection Vulnerability Pattern Detection. *Procedia Comput. Sci.* **2018**, *135*, 596–605. [CrossRef]
47. Le, V.G.; Nguyen, H.T.; Pham, D.P.; Phung, V.O.; Nguyen, N.H. GuruWS: A hybrid platform for detecting malicious web shells and web application vulnerabilities. In *Transactions on Computational Collective Intelligence XXXII*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 184–208.
48. Le, V.G.; Nguyen, H.T.; Lu, D.N.; Nguyen, N.H. A solution for automatically malicious web shell and web application vulnerability detection. In Proceedings of the International Conference on Computational Collective Intelligence, Halkidiki, Greece, 28–30 September 2016; Springer: Cham, Switzerland, 2016; pp. 367–378.
49. Wang, Z.; Yang, J.; Dai, M.; Xu, R.; Liang, X. A method of detecting Webshell based on multi-layer perception. *Acad. J. Comput. Inf. Sci.* **2019**, *2*, 81–91.
50. Cui, H.; Huang, D.; Fang, Y.; Liu, L.; Huang, C. Webshell detection based on random forest–gradient boosting decision tree algorithm. In Proceedings of the 2018 IEEE Third International Conference on Data Science in Cyberspace (DSC), Guangzhou, China, 18–21 June 2018; pp. 153–160.
51. Ai, Z.; Luktarhan, N.; Zhao, Y.; Tang, C. WS-LSMR: Malicious WebShell Detection Algorithm Based on Ensemble Learning. *IEEE Access* **2020**, *8*, 75785–75797. [CrossRef]