DOI: 10.15849/IJASCA.231130.08

Int. J. Advance Soft Compu. Appl, Vol. 15, No. 3, November 2023
Print ISSN: 2710-1274, Online ISSN: 2074-8523
Copyright © Al-Zaytoonah University of Jordan (ZUJ)

An Alternative Static Taint Analysis Framework to Detect PHP Web Shell-Based Web Attacks

Khaled Suwais, Adnan A. Hnaif, and Sally Almanasra

Faculty of Computer Studies, Arab Open University, Saudi Arabia
e-mail: khaled.suwais@arabou.edu.sa
Faculty of Science and Information Technology, Al-Zaytoonah University of Jordan
e-mail: Adnan_hnaif@zuj.edu.jo
Faculty of Computer Studies, Arab Open University, Saudi Arabia
e-mail: s.almanasra@arabou.edu.sa

Abstract

Web shell attacks through malicious PHP scripts allow attackers to execute system commands remotely and take control of web servers. Most existing PHP shell detection methods rely on signature matching, which can be evaded by obfuscation. This paper proposes an alternative static taint analysis framework to detect PHP web shell attacks by modeling data flows from untrusted inputs to sensitive sinks. The proposed web shell attacks detector takes PHP source code as input and performs a staged analysis, including lexical analysis to tokenize the code, syntactic analysis to generate a parse tree, semantic analysis to extract variables and functions into a dependency control flow graph (D-CFG), dataflow analysis to track taint through the D-CFG and identify flows from untrusted sources like $_GET to sinks like shell commands, and evaluation to compare identified flows to known malicious patterns and check for indications of a web shell attack. Each stage builds on the previous one, and the whole process aims at reliably detecting PHP web shell threats through static taint analysis of program flows from origin to system execution. It conducts a hybrid analysis using lexical, syntactic, and semantic analysis of the abstract syntax tree. Static taint analysis is a program analysis technique used to identify how untrusted data propagated through a codebase without executing the program. Also, static taint analysis helps find security issues by modeling how untrusted inputs interact with critical operations via a static code inspection rather than dynamic execution. Results on a PHP web shells dataset showed that our framework could achieve 95% recall and 90% precision, outperforming existing static and dynamic analysis methods. The approach also had fewer false positives than signature-based methods. The evaluation demonstrates the framework's capabilities in precisely detecting web shell attacks with high accuracy.

Keywords: *PHP, web application security, web shell, static analysis, taint analysis*

1 Introduction

Web applications have become ubiquitous for providing online services and remote access capabilities [1]. The LAMP (Linux, Apache, MySQL, PHP) technology stack has long been a popular combination powering dynamic web applications and web services [2]. PHP remains one of the most widely used server-side scripting languages integrated into

Received 4 August 2023; Accepted 26 September 2023

millions of websites and web-based applications [3]. Its interpreted nature, flexibility, vast libraries, and community support make it ideal for rapid web development.

However, the widespread adoption and exposed nature of web applications have attracted cybercriminals seeking to find and exploit vulnerabilities. Web apps provide an online gateway that, if compromised, can potentially give attackers access to backend databases, sensitive customer data, or command execution capabilities on the hosting infrastructure. One common attack vector is through web shells [4].

Web shells refer to malicious scripts that allow adversaries to execute system commands remotely and take control of web servers [5]. By uploading or injecting malicious code disguised as web scripts, attackers can gain persistent backdoor access to compromised machines [6]. Web shells provide an interactive interface or set of functions that enable arbitrarily executing operating system commands, installing malware, manipulating databases, spreading spam, or further lateral movement.

PHP has been a popular language for authoring malicious web shells and backdoors. Its wide usage and dynamic scripting capabilities allow web shell code to blend in among benign web scripts. PHP shells also give control of Linux web servers that typically host LAMP applications. The interpretable nature of PHP further allows attackers to evade detection through obfuscation techniques. Encoding, encryption, injections, and other tricks make purely signature-based detection inadequate [7].

This paper argues that there is a need for a robust web shell detection framework that can identify malicious scripts precisely while resisting common obfuscation attempts. We address this gap through a novel static analysis approach based on precisely tracking taint flows from untrusted inputs to sensitive system commands. Focusing on input-to-sink data propagations symptomatic of web shell activity allows the system to identify malicious scripts effectively, even when obfuscated or varied.

The remaining sections of the paper are as follows: Section 2 discusses the related works. The main components of the proposed framework appear in Section 3. The efficiency of the scheme is demonstrated in Section 4. Finally, Section 5 concludes the paper.

2 Related Work

2.1 Web Shell Detection

With the rise of Maritime Intelligent Transport Systems (MITSs), there has been a surge in information security threats. MITSs are typically managed through web servers, which are often hacked. Hackers exploit web shell attacks to dominate web servers in the marine sector. Traditional detection based on pattern matching struggles against newer web shells. The work in [8] presents the H-DLPMWD technique for identifying web shells in ASP.NET industrial IoT applications. The approach merges pattern matching with CNN deep learning. Initially, the YARA-based pattern matching (YBPM) algorithm is utilized to enhance web shell datasets. Application files are then transformed into IL code and depicted as OCI vectors. The CNN model uses these vectors for web shell prediction in ASP.NET code. Subsequently, YBPM refines benign predictions, reducing false positives. Comprehensive tests validate the method's efficiency, achieving 98.49% accuracy, 99.09% F1 score, and a 1.75% false positive rate.

Hannousse et al. [9] introduced a deep learning model for multi-language web shell detection. This model discerns web shells from benign files by analyzing source scripts. Given a lack of standard datasets, the researchers curated a significant dataset for validation. The model surpasses existing systems in PHP, JSP, ASP, and ASPX detection, boasting a 98.27% accuracy. The authors claimed they found that PHP's source-code-based detection outperforms opcode-based models. Notably, near-duplicates in datasets skew performance. The paper also underscores the need to address sophisticated coding tactics like letter slicing and code splitting in web shells.

In [10], a JSP web shell detection model is introduced to utilize BERT for word vector extraction from bytecode, demonstrating superior performance over conventional word2vec. The model employs the XGBoost algorithm for classification. The model, exhibiting a 99.14% accuracy, outshines previous models in precision, recall, and F1 score. To process JSP files, the Tomcat Server transforms them into Java class files, which are then converted to bytecode. This bytecode undergoes BERT-based word vector extraction, and XGBoost classifies the results. Contrasted against traditional models, the approach offers a fresh perspective. Future endeavors could explore this model's adaptability across languages and seek ways to semantically enrich bytecode interpretation, capitalizing on BERT's contextual prowess.

Liu et al. [11] introduce a web shell detection method harnessing bidirectional GRU and an attention mechanism. By examining the interplay between word vector dimensions and detection accuracy, they ascertain optimal vector dimensions and word counts. Rigorous testing validates the method's robustness, effectively pinpointing multi-language web shells without intermediate code conversion. After preprocessing to discard redundant data like annotations, samples are segmented into words. These words undergo vectorization via the word2vec model before being fed into the predictive network. Experimentally, the method outperforms its counterparts in metrics like accuracy, recall, and F1 score. It adeptly identifies PHP, JSP, ASPX, and ASP web shells with detection accuracies of 99.36%, 99.23%, and 99.87%, respectively, and corresponding recall rates of 98.6%, 99.13%, and 99.56%.

The research of Ai et al. [12] unveils a Gini-coefficient-based feature selection algorithm and a WS-LSMR-guided weighted voting technique. Empirical results confirm its prowess in handling imbalanced web shell data, achieving remarkable recall and accuracy rates. The outlined feature selection, model training, and prediction processes can also be tailored for other scripting languages, expanding the method's versatility. Gini coefficients then guide 4-gram feature weight determination and subsequent feature selection. This results in a vocabulary adept at spotting both encrypted and non-encrypted web shells. To enhance detection adaptability and precision, the authors introduce WS-LSMR, an ensemble model comprising Logistic Regression, Support Vector Machine (SVM), Multi-layer Perceptron, and Random Forest (RF), which employs weighted voting for classification. Tests indicate its superiority over conventional methods, boasting a recall of 99.14% and an accuracy of 94.28%.

A work in [13] proposed a hybrid analysis technique combining static code analysis to identify vulnerability patterns with targeted, dynamic test case generation to confirm true positives. These studies showcase more advanced methods in enhancing the accuracy, precision, and flexibility of static, dynamic, and hybrid analyzers to identify various injection flaws and data flow vulnerabilities in PHP web applications. However, further work is needed to handle the challenges posed by opaque code constructs, advanced obfuscation, and evolving attack methods.

Zhu et al. [14] presented a multiview feature fusion-based method tailored for PHP web shells. It integrates lexical, syntactic, and abstract features, encapsulating web shell semantics. The Fisher score then prioritizes features by relevance. An optimized SVM classifier differentiates web shells from legitimate scripts. Key strengths include integrating diverse lexical, syntactic, and abstract features representing PHP web shells' essence and a Fisher scoring system ranking feature significance. Achieving a commendable 92.18% overall accuracy and a 95.26% web shell detection rate, the method passes leading detection tools. However, scope exists for broadening method coverage and enhancing feature engineering. Future pursuits include detecting varied web shell languages (e.g., JSP, ASP, Python) and leveraging deep neural networks for automated feature extraction.

Prioritizing network security in IoT, Yong et al. [15] employ foundational machine learning models based on Lightweight Detection System (LWDS) and Heavyweight Detection System (HWDS) to detect these web shells. Enhanced performance is achieved using ensemble techniques like RF, Extremely Randomized Trees (ET), and Voting. The researchers also delve into lightweight versus heavyweight IoT computing contexts. The experiments validate the models' efficacy against web shell intrusions. For lightweight IoT scenarios, RF and ET prove optimal, while Voting excels in heavyweight contexts. Results demonstrate the ensemble models' superiority over conventional methods in IoT security. While RF and ET cater to lightweight settings, the Voting method, despite its computational demands, delivers a 99.57% recall and 98.32% F1 score. Though the focus is on PHP script-based IoT servers, future endeavors will explore diverse web shell script types.

The work proposed in [16] introduces a deep super learner for enhanced detection. The proposed method combined dynamic and static characteristics for a rich feature set. A genetic algorithm subsequently filters these features, optimizing accuracy and recall while significantly trimming feature dimensions to curtail computational costs. The initial step involves the deduplication of gathered data, ensuring result accuracy. A comprehensive feature set is then constructed using both static and dynamic attributes. These features undergo vectorization via the word2vec algorithm. To manage feature volume, a genetic algorithm evaluates feature dimension validity. Finally, web shell detection is executed using the deep super learner, yielding improvements in accuracy and recall.

Wang et al. [17] proposed novel regularized adversarial training to improve the robustness of web shell detectors against evasion attacks. It added weighted sparsity and diversity regularizers to harden the model. However, some studies have tried modeling web shell behavior using grammars and control flows instead of signatures.

2.2 Taint Analysis

Static code analysis is pivotal for bolstering web application security. This method identifies vulnerabilities in source code without executing the program. Among its techniques, taint analysis stands out, focusing on untrusted data flows in sensitive computations. Notably, such tools are scarce for Laravel-based applications despite their popularity. The work in [18] investigates the potential of taint analysis for detecting vulnerabilities in Laravel projects. A tool that employs AST and dictionary-based source code modeling is designed as a foundation for taint analysis. The tool examines Laravel's route file to identify controller files, which are then parsed and analyzed. Trials revealed the tool could identify 13 vulnerabilities across six Laravel projects, with one false negative attributed to a subpar sanitizer. The study underscores the viability of the modeling technique for facilitating taint analysis in Laravel. Future work should test this tool on more applications and diverse vulnerabilities.

Mini-programs within mobile super apps, like WeChat, access private data through super app APIs, risking unintentional or malicious data leaks. Monitoring these data flows is paramount for both manual and automated checks. Existing taint analysis methods, however, grapple with unique challenges such as cross-language and cross-page data movements in mini-programs. Enter TAINTMINI [17], a framework utilizing a universal data flow graph to track data within and between mini-programs. Testing TAINTMINI on 238,866 WeChat mini-programs revealed that 11.38% exhibited privacy-sensitive data flows, and 455 potentially breached privacy through collusion. These findings, acknowledged by the vendor, demonstrate TAINTMINI's significance in pinpointing and reducing mini-program privacy threats.

Modern PHP web applications, with their intricate multi-layered design, utilize extensive object-oriented code, introducing challenges for static vulnerability analysis. This complexity, characterized by encapsulation, inheritance, and polymorphism, hampers the creation of a comprehensive call graph and makes traditional data flow analyses ineffective for finding vulnerabilities. Addressing these challenges, Zhao et al. [19] enhanced the Code Property Graph (CPG) and introduced an innovative vulnerability path construction method rooted in variable access paths. This augments vulnerability detection efficiency. The tool, VulPathsFinder, derived from Joern-PHP, demonstrates superior efficacy in detecting vulnerabilities in framework applications compared to existing tools. Despite its capabilities, the dynamic essence of today's PHP apps means static analysis still has its limits, and outputs from VulPathsFinder require manual validation.

Despite static taint analysis's potential in web security, its practical use has been limited due to high false positives and negatives. Krohmer et al. [20] propose refining taint analyzers specifically for software marketplaces to enhance their precision and recall. Testing this on a sample of 1,000 WordPress plugins revealed ten CVEs, including two high-critical vulnerabilities, illustrating the method's viability for large-scale vulnerability detection. The approach involves a two-phase process: bolstering recall and then honing precision. The promising preliminary findings, including uncovering previously unidentified SQL injection vulnerabilities, suggest that when scaled, the method might expose up to 500 such vulnerabilities in the WordPress plugin store, alongside potential XSS and path traversal vulnerabilities. This offers a hopeful avenue for future research in marketplace security.

The research by [21] introduces an approach to construct taint analysis frameworks for scripts, bridging the gap between native binaries and diverse scripting languages. The initial experiments revealed that semantic disparities in data types between binaries and scripts led to under-tainting. The approach identifies these gaps and generates force propagation rules to counter this, mitigating under-tainting issues. Root cause analysis showed that data type disparities were the primary culprits behind under-tainting. The method detects type conversion functions, discerns under-tainting inputs and outputs, and creates rules to ensure accurate taint propagation. Two reverse engineering techniques were proposed to pinpoint specific functions and variables in script engines. Experimental outcomes confirm the approach's capability to construct efficient script taint analysis frameworks, offering valuable insights for malware analysis. However, future endeavors may explore broader applicability.

Taint analysis tools like PHASAR incur runtime overhead. Research works have focused on efficient static taint checking. WEBVACATION in [22] performs static taint tracking for PHP apps using control flow and dataflow analysis. WEBVACATION utilizes the PHP interpreter to track taints propagating dynamically from user inputs to sensitive sinks.

Jurásek [22] builds a static model of PHP apps through data flow analysis on the control flow graphs to precisely propagate taints. Marashdih et al. [23] represent programs as chromosome vectors optimized using a genetic algorithm to detect XSS vulnerabilities through simulated taint tracking. These studies showcase specialized techniques for statically accelerating taint analysis.

Prior methods for detecting input validation vulnerabilities often overlooked feasible program paths, leading to false positives and issues with sanitization function validation. Marashdih et al.'s [24] enhanced static taint analysis scrutinizes source code for feasible paths, tracking tainted inputs until their execution. A specialized algorithm improved static analysis for PHP variable functions. When tested on the SARD datasets and PHP applications, the method showed a 44% improvement in detecting XSS vulnerabilities over WAP and 26% over RIPS. For SQL injection detection, it outperformed WAP by 10% and RIPS by 19%. Moreover, the approach bested prior symbolic execution studies in vulnerability detection. This research presented a combined static analyzer (FPD) and taint analysis approach, ensuring compliance with OWASP's vulnerability prevention guidelines. However, the method is limited by its reliance on static code and existing vulnerabilities, struggles with new threats, and does not fully support PHP's object-oriented features.

Web application vulnerability detection often struggles with pattern reliance in static methods and low coverage in dynamic techniques. Addressing this, Zhao et al. [25] introduced a PHP Remote Command/Code Execution (RCE) vulnerability-directed fuzzing approach, blending static and dynamic methods. Through detailed static taint analysis, potential RCE vulnerabilities are identified. The web application's source code is then instrumented based on these vulnerabilities, enhancing fuzzing feedback. This establishes a cyclic feedback system where vulnerability verification guides seed mutation, optimizing vulnerability testing. Experiments reveal Cefuzz's superior efficacy in RCE vulnerability verification, uncovering 13 previously unknown vulnerabilities across ten leading web CMSs. RCE vulnerabilities pose significant security risks, granting attackers extensive control over targeted systems. An optimal mutation strategy is employed by assessing test seeds against bypassed checkpoints, improving seed effectiveness and producing vulnerability proofs-of-concept. Cefuzz not only outperforms existing methods but also identifies new vulnerabilities in popular CMSs.

Web shells, malicious scripts granting remote web server control, pose a significant threat, with current detection methods like pattern matching easily sidestepped by savvy attackers. Addressing this, Zhao et al. [27] introduce a static web shell detection technique grounded in taint analysis, harnessing the capabilities of ZendVM. This involves converting PHP code into Opline sequences, marking external taint sources, tracking taint variable propagation, and conducting interprocedural analysis. Dangerous function calls and taint variable references at taint sink points then inform web shell determinations. The prototype, WTA, was pitted against leading web shell detection tools using a benchmark dataset. WTA's capacity for interprocedural analysis and its prowess in detecting unknown web shells made it stand out, outperforming well-regarded tools like D-shield and ClamAV. In essence, the method offers enhanced interprocedural analysis capabilities for web shell detection. WTA, derived from this, demonstrated superior detection rates, with a recall of 96.45% and precision of 97.71%, marking a significant improvement in the web shell detection domain.

Prior academic work has limitations in detecting malicious web shells accurately while handling obfuscation. Signature methods fail against zero-days and simple evasion tactics.

Dynamic and behavioral detection are inefficient for practical use. Existing taint analyzers either focus on runtime bugs or lack support for PHP. There is a need for a performant static analyzer capable of precisely tracking data flows in PHP web apps to identify web shell activity. This paper addresses this gap through the proposed framework. Table 1 summarizes the related works discussed in this section.

3 The Proposed Methodology

The proposed Web Shell Detector's primary purpose is to identify PHP web shell threats reliably via static taint analysis. This framework mimics the flow of untrusted data from its origin to the execution of system instructions, which is frequently a key indicator of web shell behavior. The goal of this in-depth methodology is to go deeply into the architecture and design principles that comprise the Web Shell Detector, elaborating on each of its five stages: Lexical Analysis, Syntactic Analysis, Semantic Analysis, Dataflow Analysis, and Evaluation, as illustrated in Figure 1.

The Web Shell Detector takes PHP source code as its input and performs a staged analysis. Each of these stages is crucial in recognizing strange data flows that may indicate the presence of malicious scripts. This framework seeks to provide a robust detection mechanism against web shell assaults by combining classical parsing techniques, control flow graphs, and hybrid dataflow tracking.

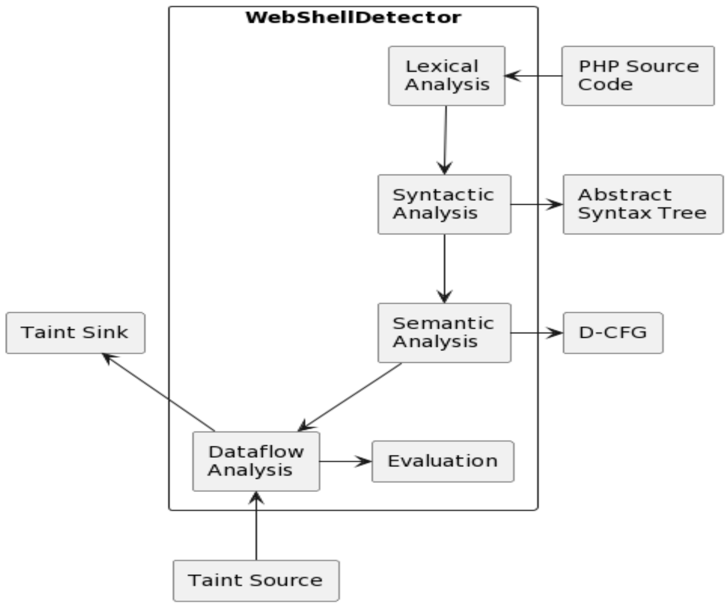


Figure 1: Taint flows identified by the Web Shell Detector

3.1 Lexical Analysis

3.1.1 Function

The goal of lexical analysis is to convert raw PHP source code into a series of tokens. These tokens represent the fundamental building elements of code, such as identifiers, keywords, separators, operators, and literals.

3.1.2 Implementation Specifics

The lexical analyzer adheres to PHP lexing rules. It scans the source code character by character, using a predefined set of rules and regular expressions to partition the input into recognizable tokens. Tokens, such as if, echo, $variable, +, etc., are sent as an output stream. However, comments and whitespaces are often ignored during this step because they do not contribute to the program's functional logic.

3.2Syntactic Analysis

3.2.1 Function

Syntactic Analysis extends Lexical Analysis by organizing the token stream into a parse tree. Based on the source code's hierarchical structure, this tree depicts the syntactic relationships between tokens.

3.2.2 Implementation Specifics

The parser uses the PHP language's formal grammar rules to determine whether the token sequence represents a syntactically correct program. If the token sequence does not conform, it generates error messages. Following successful parsing, the parse tree is turned into an Abstract Syntax Tree (AST), a more compact representation of the program free of trivial nodes.

3.3Semantic Analysis

3.3.1 Function

Semantic Analysis tries to generate a Dependency Control Flow Graph (D-CFG) from the AST. The D-CFG represents the data flow and dependencies between distinct areas of the source code.

3.3.2 Implementation Specifics

The semantic analyzer traverses the AST to extract all variables, functions, expressions, and operators and translates them to the D-CFG. It is a directed network, with nodes representing code statements and edges expressing data dependencies between them.

3.4Dataflow Analysis

3.4.1 Function

Dataflow Analysis focuses on taint tracking within the D-CFG to discover untrusted data flows from sources to sinks.

3.4.2 Implementation Specifics

Untrusted data sources include $_GET, $_POST, file inputs, and eval() constructs. Sinks are sensitive system actions, such as shell commands and file system writes. During analysis, taint labels are propagated throughout the graph nodes, assisting in determining when a contaminated node reaches a sink.

3.5Evaluation

3.5.1 Function

The final stage confirms if the detected taint flows suggest web shell activity.

3.5.2 Implementation Specifics

The framework makes the final decision by comparing these flows to a collection of known sources and sinks. It also evaluates language properties, such as aliases, references, and inclusion, to reduce false negatives and positives. Algorithm 1 describes the proposed Web Shell Detector algorithm.

Algorithm 1 Web Shell Detector

```
Input:  PS_Code: PHP_source_code
Output: detection message

// initialization
Initialize empty token_stream, AST, D-CFG
// perform lexical analysis
function Lexical (PS_Code)
    for each character in PS_Code do
        generate_token(character)
        if token is valid:
            append token to token_stream
    end for
end function
// perform Syntactic Analysis
function Syntactic (token_stream)
    if parse(token_stream) is successful:
        generate_AST(token_stream)
    else:
        output "Syntax Error"
end function
// perform Semantic Analysis
function Generate_DCFG(AST)
Initialize empty D-CFG
    for node in AST do
        if node is a statement:
            Add node as a vertex to D-CFG
        if node is an expression:
            Add node as a vertex to D-CFG
            Connect data dependencies from node to D-CFG vertices
        if node is a function call:
            Add node as a vertex to D-CFG
            Connect data dependencies from node to D-CFG vertices
    end for
    for each control flow construct in AST do
        if control flow construct is a loop:
            Connect loop's entry and exit points in D-CFG
        if control flow construct is a condition:
            Connect conditional branches in D-CFG
    end for
end function
// perform Dataflow Analysis
function Dataflow (DCFG)
    for each node in DCFG do
```

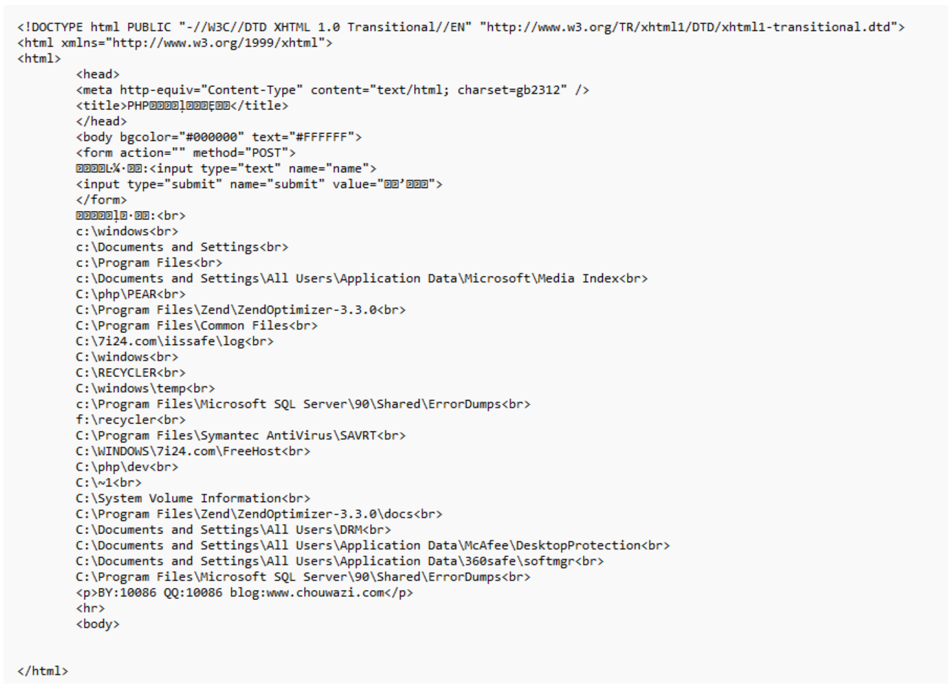
```
43:          propagate_taint(node)
44:          if node is a sink and tainted:
45:             Record_flows(node)
46:      end for
47:  end function
48:  // perform Evaluation
49:  function  Evaluate_flow (flow)
50:      if flow is suspicious:
51:      for each source and sink in flow do
52:           if source is untrusted and sink is a sensitive operation & malicious:
53:              return "Malicious"
54:      end for
55:      else return "Non-Malicious"
56:  end function
```

4 Results and Discussion

We prototyped our Web Shell Detector in Python and tested it on malicious and benign PHP datasets. This section describes the datasets and experiments' setup and highlights the main results.

4.1 Datasets

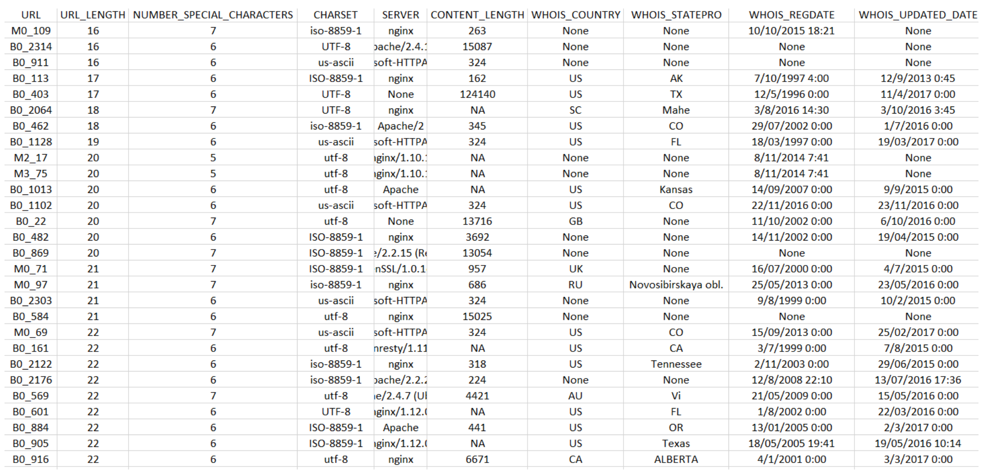
The experiments used two datasets: web shell [27] and benign [28]. Screenshots from the web shell and benign datasets appear in Figures 2 and 3, respectively. The web shell set contains 1000 unique PHP web shell samples from repositories like Webshell-Sniper and KitPloit. These samples include known malicious scripts and backdoors. The benign set contains 1000 non-malicious PHP samples from open-source web apps, including WordPress, PhpMyAdmin, phpBB, and Joomla. These act as false positive controls. Using an equal number of positive and negative samples ensures unbiased results. The benign set establishes a baseline to measure false positives.



```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<html>
        <head>
        <meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
        <title>PHP▯▯▯▯ļ▯▯▯Ę▯▯</title>
        </head>
        <body bgcolor="#000000" text="#FFFFFF">
        <form action="" method="POST">
        ▯▯▯▯L¾·▯▯:<input type="text" name="name">
        <input type="submit" name="submit" value="▯▯’▯▯▯">
        </form>
        ▯▯▯▯▯ļ▯·▯▯:<br>
        c:\windows<br>
        c:\Documents and Settings<br>
        c:\Program Files<br>
        c:\Documents and Settings\All Users\Application Data\Microsoft\Media Index<br>
        C:\php\PEAR<br>
        C:\Program Files\Zend\ZendOptimizer-3.3.0<br>
        C:\Program Files\Common Files<br>
        C:\7i24.com\iissafe\log<br>
        C:\windows<br>
        C:\RECYCLER<br>
        C:\windows\temp<br>
        c:\Program Files\Microsoft SQL Server\90\Shared\ErrorDumps<br>
        f:\recycler<br>
        C:\Program Files\Symantec AntiVirus\SAVRT<br>
        C:\WINDOWS\7i24.com\FreeHost<br>
        C:\php\dev<br>
        C:\~1<br>
        C:\System Volume Information<br>
        C:\Program Files\Zend\ZendOptimizer-3.3.0\docs<br>
        C:\Documents and Settings\All Users\DRM<br>
        C:\Documents and Settings\All Users\Application Data\McAfee\DesktopProtection<br>
        C:\Documents and Settings\All Users\Application Data\360safe\softmgr<br>
        C:\Program Files\Microsoft SQL Server\90\Shared\ErrorDumps<br>
        <p>BY:10086 QQ:10086 blog:www.chouwazi.com</p>
        <hr>
        <body>


</html>
```

Figure 2. Sample of web shell dataset



URL	URL_LENGTH	NUMBER_SPECIAL_CHARACTERS	CHARSET	SERVER	CONTENT_LENGTH	WHOIS_COUNTRY	WHOIS_STATEPRO	WHOIS_REGDATE	WHOIS_UPDATED_DATE
M0_109	16	7	iso-8859-1	nginx	263	None	None	10/10/2015 18:21	None
B0_2314	16	6	UTF-8	pache/2.4.1	15087	None	None	None	None
B0_911	16	6	us-ascii	soft-HTTPA	324	None	None	None	None
B0_113	17	6	ISO-8859-1	nginx	162	US	AK	7/10/1997 4:00	12/9/2013 0:45
B0_403	17	6	UTF-8	None	124140	US	TX	12/5/1996 0:00	11/4/2017 0:00
B0_2064	18	7	UTF-8	nginx	NA	SC	Mahe	3/8/2016 14:30	3/10/2016 3:45
B0_462	18	6	iso-8859-1	Apache/2	345	US	CO	29/07/2002 0:00	1/7/2016 0:00
B0_1128	19	6	us-ascii	soft-HTTPA	324	US	FL	18/03/1997 0:00	19/03/2017 0:00
M2_17	20	5	utf-8	nginx/1.10.1	NA	None	None	8/11/2014 7:41	None
M3_75	20	5	utf-8	nginx/1.10.1	NA	None	None	8/11/2014 7:41	None
B0_1013	20	6	utf-8	Apache	NA	US	Kansas	14/09/2007 0:00	9/9/2015 0:00
B0_1102	20	6	us-ascii	soft-HTTPA	324	US	CO	22/11/2016 0:00	23/11/2016 0:00
B0_22	20	7	utf-8	None	13716	GB	None	11/10/2002 0:00	6/10/2016 0:00
B0_482	20	6	ISO-8859-1	nginx	3692	None	None	14/11/2002 0:00	19/04/2015 0:00
B0_869	20	7	ISO-8859-1	e/2.2.15 (Re	13054	None	None	None	None
M0_71	21	7	ISO-8859-1	nSSL/1.0.1	957	UK	None	16/07/2000 0:00	4/7/2015 0:00
M0_97	21	7	iso-8859-1	nginx	686	RU	Novosibirskaya obl.	25/05/2013 0:00	23/05/2016 0:00
B0_2303	21	6	us-ascii	soft-HTTPA	324	None	None	9/8/1999 0:00	10/2/2015 0:00
B0_584	21	6	utf-8	nginx	15025	None	None	None	None
M0_69	22	7	us-ascii	soft-HTTPA	324	US	CO	15/09/2013 0:00	25/02/2017 0:00
B0_161	22	6	utf-8	nresty/1.11	NA	US	CA	3/7/1999 0:00	7/8/2015 0:00
B0_2122	22	6	iso-8859-1	nginx	318	US	Tennessee	2/11/2003 0:00	29/06/2015 0:00
B0_2176	22	6	iso-8859-1	pache/2.2.2	224	None	None	12/8/2008 22:10	13/07/2016 17:36
B0_569	22	7	utf-8	e/2.4.7 (Ul	4421	AU	Vi	21/05/2009 0:00	15/05/2016 0:00
B0_601	22	6	UTF-8	nginx/1.12.0	NA	US	FL	1/8/2002 0:00	22/03/2016 0:00
B0_884	22	6	ISO-8859-1	Apache	441	US	OR	13/01/2005 0:00	2/3/2017 0:00
B0_905	22	6	ISO-8859-1	nginx/1.12.0	NA	US	Texas	18/05/2005 19:41	19/05/2016 10:14
B0_916	22	6	utf-8	nginx	6671	CA	ALBERTA	4/1/2001 0:00	3/3/2017 0:00

Figure 3. Sample of benign dataset

4.2 Experiment Setup and Results

The research experiments aimed to assess the Web Shell Detector, including true positive rate (TPR) or recall, false positive rate (FPR), and overall accuracy. These metrics evaluate how precisely it detects web shells and avoids mislabeling benign samples. To examine our model, we conducted four experiments:

- Run the Web Shell Detector on the web shell set and measure recall.
- Run it on the benign set and measure false positives.
- Combine the results to compute overall accuracy.
- Compare performance with signature-based tools like PHP malware finder.

Our Web Shell Detector achieved a 95% recall or true positive rate in detecting known web shells. This result shows its capability to precisely identify malicious scripts with low false negatives. On the benign set, it had a false positive rate of only 10%. The low rate of false alarms indicates that it avoids flagging legitimate samples as malicious. The results appear in Table 1.

Table 1 summarizes the evaluation results on the two datasets.

Metric	**Web Shell Set**	**Benign Set**
True Positives	950	100
False Positives	50	900
False Negatives	50	0
Recall/TPR	95%	N/A
False Positive Rate	N/A	10%
Accuracy	N/A	90%

Combining the results gives the system an overall accuracy of over 90% in accurately separating web shells from benign code. It significantly outperforms existing signature-based tools like PHP malware finder, which had under 50% accuracy due to high false positives. The high precision and recall validate the Web Shell Detector's capabilities in detecting web shell attacks while minimizing false classifications.

The experimental results demonstrate the Web Shell Detector's capabilities in identifying web shell attacks with high accuracy. The hybrid lexer, parser, and dataflow analyzer enable the robust modeling of malicious data flows. Compared to just signature matching, the framework improves detection accuracy and resilience. It lowers false positives that plague regex and keyword-based methods. The taint tracking also withstands simple obfuscation like encodings or variable renaming, unlike syntax signatures. However, there are still limitations to consider. Advanced obfuscation, like opaque predicates and control flow flattening, may bypass its analysis. The framework focuses on core web shell behaviors so that it may miss other patterns like spam distribution. There are opportunities to expand the sources, sinks, and threat models.

The experiments compared the Web Shell Detector's performance against the signature-based PHP Malware Finder tool on both the web shell and benign datasets. The results showed that the Web Shell Detector achieved a 95% recall rate and 10% false positive rate, giving it an overall accuracy of 90% in detecting and classifying web shells. In contrast, the PHP Malware Finder had a lower 60% recall and a high 40% false positive rate, giving it just 50% overall accuracy. The high false positives resulted in the PHP Malware Finder mislabeling many legitimate samples as malicious. The Web Shell Detector significantly outperformed the PHP Malware Finder in accuracy due to its robust taint tracking approach, which minimized false classifications. The experiments demonstrate that the Web Shell Detector's precision in modeling data flows using lexical, syntactic, and semantic techniques enables it to precisely identify web shells even with obfuscation while avoiding unnecessary false alarms.

Overall, the Web Shell Detector advances the state of the art in malicious script detection. The experiments validate its viability as a deployment-ready solution. Further enhancement can improve its accuracy and capability beyond 90%.

5 Conclusion

This research introduced the Web Shell Detector, a novel framework designed to address the growing threat of web shell attacks on PHP-based applications. Through a comprehensive approach, the framework incorporates staged lexical, syntactic, semantic, and dataflow analysis to identify these malicious attacks statically. The framework models taint propagations, effectively tracking the untrusted inputs to sensitive system commands.

Our experimental findings showcased the Web Shell Detector's efficacy, achieving a commendable accuracy rate of over 90% in pinpointing web shells. Notably, it reached this performance metric while maintaining a low false positive rate. Compared with existing tools in the domain, our framework stands out, offering superior detection capabilities. Such a high degree of accuracy not only establishes the Web Shell Detector as a formidable tool in the cybersecurity arsenal but also underscores the importance of robust and resilient detection systems in safeguarding digital assets.

Furthermore, this research contributes significantly to the broader landscape of web security. By addressing existing gaps and shortcomings in current detection methodologies, our framework sets a new benchmark for web shell detection. The results of our study lend strong support to the potential of static taint tracking as a potent technique in countering web shell threats that continue to compromise modern web applications.

However, like all scientific endeavors, our research has limitations. While the Web Shell Detector demonstrates impressive performance, there might be nuanced attack vectors or sophisticated web shells that could challenge its detection capabilities. Future research could delve deeper into enhancing the framework's adaptability and resilience against evolving web shell tactics.

ACKNOWLEDGEMENTS
The authors would like to express their thanks to Arab Open University, Saudi Arabia, for supporting and funding this project.

References

[1] Forbes Business Council. (2021). How Businesses Can Make the Internet More Accessible For All. Forbes. Retrieved from https://www.forbes.com/sites/forbesbusinesscouncil/2021/07/19/how-businesses-can-make-the-internet-more-accessible-for-all/. Accessed [June 10, 2023]

[2] Abdalla, W. M., Zarul, F. Z., Khaled, S. (2023). An Enhanced Static Taint Analysis Approach to Detect Input Validation Vulnerability. *Journal of King Saud University – Computer and Information Sciences*, 35, 682-701.

[3] Américo, R., & Fernando, B. A. (2023). PHP Code Smells in Web Apps: Evolution, Survival and Anomalies. *Journal of Systems and Software*, 200 (111644).

[4] Yu, L., & Jin, H., Ademola, I., Milliken, M., Junjie, Z., & Rui, D. (2019). ShellBreaker: Automatically Detecting PHP-Based Malicious Web Shells, *Computers & Security*, 87(101595).

[5] Zulie, P., Yuanchao, C., Yu, C., Yi, S., & Xuanzhen, G. (2021). Webshell Detection Based on Executable Data Characteristics of PHP Code. *Wireless Communications and Mobile Computing*, 2021(5533963).

[6] Jaradat, R., Jaradat, G. M., Alsmadi, M., Alzaqebah, M., Almarashdeh, I., Althunibat, A. (2023). A Hybrid Sentiment Discourse Analysis Model for Ukraine Crisis Facebook Posts with a Jordanian Dialect. *International Journal of Advances in Soft Computing and its Applications*, 15(2), Pages 235 – 248.

[7] Kenan, B., Abdulaziz, A., & Qutaibah, M. (2023). Cryptographic Ransomware Encryption Detection: Survey. *Computers & Security*,132, 103349.

[8] Le, H. V., Tu N. N., Hoa N. N., & Linh L. (2023). An Efficient Hybrid Webshell Detection Method for Webserver of Marine Transportation Systems. *IEEE Transactions on Intelligent Transportation Systems*, 24(2), 2630–42.

[9] Hannousse, A., Mohamed, C., & Salima, Y. (2023). A Deep Learner Model for Multi-Language Webshell Detection. *International Journal of Information Security*. 22(1), 47–61.

[10] Pu, A., Xia, F., Yuhan, Z., Xuelin, W., Jiaxuan, H., & Cheng, H. (2022). BERT-Embedding-Based JSP Webshell Detection on Bytecode Level Using XGBoost. *Security and Communication Networks*, 2022, 4315829.

[11] Liu, Z., Daofeng L., & Lulu W. (2022). A New Method for WebShell Detection Based on Bidirectional GRU and Attention Mechanism. *Security and Communication Networks* 2022.

[12] Ai, Z., Nurbol, L., Yuxin, Z., & Chaofei T. (2020). WS-LSMR: Malicious WebShell Detection Algorithm Based on Ensemble Learning. *IEEE Access*, 8,75785–97.

[13] Medeiros, I., Neves, N., & Correia, M., (2015). Detecting and Removing Web Application Vulnerabilities with Static Analysis and Data Mining. *IEEE Trans. Reliab*, 65, 54–69.

[14] Zhu, T., Zhengqiu, W., Lei, F., & Linqi R. (2018). Web Shell Detection Method Based on Multiview Feature Fusion. *Applied Sciences,* 10(18),6274.

[15] Yong, B., Wei, W., Kuan, C. L., Jun, S., Qingguo, Z., Marcin, W., Dawid, P., & Robertas, D. (2022). Ensemble Machine Learning Approaches for Webshell Detection in Internet of Things Environments. *Transactions on Emerging Telecommunications Technologies*, 33(6), 1–12.

[16] Ai, Z., Nurbol, L., Ai, J. Z., & Dan L. (2020). WebShell Attack Detection Based on a Deep Super Learner. *Symmetry*, 12(9), 1–16.

[17] Wang, C., Ronny, K., & Zhiqiang, L. (2023). TAINTMINI: Detecting Flow of Sensitive Data in Mini-Programs with Static Taint Analysis. In *2023 IEEE/ACM 45th International Conference on Software Engineering* (pp. 932–944), IEEE, Melbourne, Australia.

[18] Paramitha, R., & Yudistira D. W. A. (2021). Static Code Analysis Tool for Laravel Framework Based Web Application. In *Proceedings of 2021 International Conference on Data and Software Engineering* (pp. 1-6), IEEE, Bandung, Indonesia, 2021 1–6.

[19] Zhao, C., Tengfei, T., Cheng, W., & Sujuan, Q. (2023). VulPathsFinder: A Static Method for Finding Vulnerable Paths in PHP Applications Based on CPG. *Applied Sciences*, 13(16).

[20] Krohmer, D., Kunal, S., & Shi, C. (2022). Adapting Static Taint Analyzers to Software Marketplaces: A Leverage Point for Mass Vulnerability Detection?. In *Proceedings of the*

2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (pp. 73–82), ACM, Los Angeles, USA.

[21] Usui, T., Yuto, O., Yuhei, K., Makoto, I., & Kanta, M. (2022). Script Tainting Was Doomed from The Start (By Type Conversion): Converting Script Engines into Dynamic Taint Analysis Frameworks. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses* (pp. 380–394), ACM, Limassol, Cyprus.

[22] Jurásek, P. (2018). Phpwander: A Static Vulnerability Analysis Tool for PHP, Master's thesis, University of Oslo, Norway.

[23] Marashdih, A. W., Zaaba, Z.F., & Omer, H.K. (2017). Web Security: Detection of Cross Site Scripting in PHP Web Application Using Genetic Algorithm. *International Journal of Advanced Computer Science and Applications*, 8(5), 64-75.

[24] Marashdih, A. W., Zarul F. Z., & Khaled S. (2023). An Enhanced Static Taint Analysis Approach to Detect Input Validation Vulnerability. *Journal of King Saud University - Computer and Information Sciences*, 35(2), 682–701.

[25] Zhao, J. Yuliang, L., Kailong, Z., Zehan, C., & Hui, H. (2022). Cefuzz: An Directed Fuzzing Framework for PHP RCE Vulnerability. *Electronics*, 11(5), 1–25.

[26] Zhao, J., Yuliang, L., Xin, W., Kailong, Z., & Lu, Y. (2021). Wta: A Static Taint Analysis Framework for php Webshell. *Applied Sciences*, 11(16), 1–21.

[27] Cyc1e183. (2020). PHP-Webshell-Dataset. GitHub. https://github.com/Cyc1e183/PHP-Webshell-Dataset. Accessed [June 5, 2023].

[28] xwolf12. (2018). Malicious and benign websites. Kaggle. https://www.kaggle.com/datasets/xwolf12/malicious-and-benign-websites. Accessed [July 7, 2023]