applied sciences

MDPI

Article

Cross-Site Scripting Guardian: A Static XSS Detector Based on Data Stream Input-Output Association Mining

Chenghao Li [1,†], Yiding Wang [1,†], Changwei Miao [1,†] and Cheng Huang [1,2,*]

[1] College of Cybersecurity, Sichuan University, Chengdu 610064, China; 2017141531008@stu.scu.edu.cn (C.L.); 2016141043006@stu.scu.edu.cn (Y.W.); 2017141472035@stu.scu.edu.cn (C.M.)

[2] Guangxi Key Laboratory of Cryptography and Information Security, Guilin 541004, China

* Correspondence: codesec@scu.edu.cn

† These authors contributed equally to this work.

Received: 7 June 2020; Accepted: 2 July 2020; Published: 9 July 2020

Abstract: The largest number of cybersecurity attacks is on web applications, in which Cross-Site Scripting (XSS) is the most popular way. The code audit is the main method to avoid the damage of XSS at the source code level. However, there are numerous limits implementing manual audits and rule-based audit tools. In the age of big data, it is a new research field to assist the manual auditing through machine learning. In this paper, we propose a new way to audit the XSS vulnerability in PHP source code snippets based on a PHP code parsing tool and the machine learning algorithm. We analyzed the operation sequence of source code and built a model to acquire the information that is most closely related to the XSS attack in the data stream. The method proposed can significantly improve the recall rate of vulnerability samples. Compared with related audit methods, our method has high reusability and excellent performance. Our classification model achieved an F1 score of 0.92, a recall rate of 0.98 (vulnerable sample), and an area under curve (AUC) of 0.97 on the test dataset.

Keywords: vulnerability detection; code audit; cross-site scripting; machine learning

1. Introduction

With the rise of communication technology such as 5G technology, data transmission ability has been greatly improved and web technology has been more widely used. The browser and server (B&S) architecture is highly convenient in that it can free the steps of the installation. Thus, applications of B&S architecture are constantly developing. Predictably, more and more services will tend to be part of a Web application. However, it is well known that attacks on web applications represent the largest number of threats and this will take on more network attacks. Due to the lack of security awareness and the around usage of Web Application Firewall (WAF) technology, it is common for developers to ignore the vulnerability on the source code level. Although WAF technology can usually intercept a considerable number of malicious attacks, a secure Web application should not rely solely on WAF technology to secure it [1]. If the source code is not reasonably modified, threats will always exist. In fact, according to Web Applications vulnerabilities and threats statistics for 2019, 82% of the vulnerabilities are in application code, which indicates the vulnerability audit itself is not negligible [2].

A traditional code audit is usually done manually, which leads to large human resources consumption and high false rates. Thus, several automatic auditing tools have been developed to assist auditors. However, a great number of current tools are matched by a large library of rules and regulations which indicates they are somehow lacking of flexibility. With the increasing demand for data analysis, the efficient acquisition of knowledge through machine learning has gradually become the main driving force. "How to make a deep analysis of complex and diverse data" becomes the main

direction of research [3]. At present, researches on code vectorization make it possible for machine learning technology to be applied in a code audit [4].

Contributions

We assume that machine learning model can learn to mine the specific association between the filters and the I/O context to prevent XSS attacks. Then, we can detect XSS vulnerabilities in PHP source code based on this association.

In summary, the contributions of this paper are as follows.

(1) **Cross-Scripting Guardian**: A novel PHP source code vulnerability detector. According to our experimental evaluation, this detector performs better in vulnerability identification compared with related methods. It has guiding significance for the follow-up research of machine learning applied to XSS code audits.
(2) **Algorithms for input-output pattern mining**: We analyzed the I/O patterns in XSS-related PHP source code and design an algorithm to identify the specific path of the data stream. With this algorithm, it will be more efficient to mine the I/O context and build the precise and consistent TOKEN sequence for representation.
(3) **A novel adapted open-sourced PHP code parsing tool** [5]: We modify the source code of the VLD to enable it to output richer and more structured parsing information of the PHP source code, which can facilitate the work of parsing and serializing PHP source code.

The following sections are organized as follows: in Section 2, we introduce the background of the research, the current work, and results related to the code audit. In Section 3, we develop the approaches and theoretical methods of the framework. In Section 4, the experimental goal, process, results, and evaluation are presented. In the last section, we summarize our work and list several prospects for future work.

2. Research Background and Significance

2.1. Background

According to OWASP (Open Web Application Security Project) data [6], XSS vulnerabilities have been top-10 among all Web application vulnerabilities for years, while nearly 80% of Web application servers have been built with PHP.

Nowadays, SQL injection has become increasingly tight. XSS attacks are the preferred way for more hackers to exploit vulnerabilities. It is highly harmful, has a wide range of influence, and can be combined with other forms of attack, which indicates that the prevention of XSS attacks in Web applications is urgent.

With the development of 5G technology, the ability of data transfer will be further improved. As the Web application has entered the third generation [7], more and more services will be carried out in the form of a Web application soon. XSS is one of the most common approaches of attack in Web applications, and the corresponding defense measures should be further emphasized [8]. Although defense against XSS has become an important part of WAF, a robust Web application should not rely solely on an external firewall to ensure security. This security threat will never be eliminated without fixing the corrupted code portion. Therefore, auditing the vulnerability of Web applications from the source code level is an essential part and a new research direction of cybersecurity. Considering the popularity of the PHP language in Web deployment, our research revolves around PHP code.

2.2. Related Work

2.2.1. Code Auditing with Machine Learning

In the past years, code auditing in web applications was mostly done manually, making it difficult to ensure efficiency and accuracy. It wastes more time and manpower facing large-scale projects [9]. Therefore, several tools have been developed to help and supplement the audit work. However, most of the current PHP code vulnerability auditing tools are based on the predetermined detection rules. They match the code blocks by using the regular matching through a huge rule base, which has a large improvement space [10]. With the development of code vectorization, auditing models of machine learning are proposed and optimized [11,12].

AST&CFG: Code vectorization is prior research on machine learning-based code auditing. Abstract Syntax Tree (AST) and Control Flow Graph (CFG) are the most popular technologies and classic code analysis methods. They abstract code logic and give serialization results, which are quite suitable for the input of machine learning models. The approach of code vectorization based on AST and CFG was first proposed by Michael et al. [13]. These two special data structures represent the source code and eliminate the noise caused by the different code styles of programmers. However, the drawback is that they all complicate the original code structure into another data structure. The new complicated data structure requires additional methods to parse, which will undoubtedly increase the workload of researchers. To some extent, it may lose some important information due to different parsing methods.

Based on this approach, auditing tools, such as RIPS [14], Cobra [15], etc., proposed code-based context detection ways. RIPS focuses on the tracking and analysis of sensitive function calls and data stream. Cobra parses the source code into AST based on lexical analysis, then determines whether there are vulnerabilities according to the controllability of the parameters of sensitive functions. However, these mainstream tools have a high rate of false positives, i.e., they misreport relatively more safe code blocks as vulnerabilities. In our tests, Cobra only found less than 50 XSS vulnerabilities in the results of scanning the WordPress project (version 1.5) [16], while the commercial version of Fortify found 450.

Tokenization: With the development of Natural Language Processing (NLP), tokenization of code and sequence analysis has become a new direction in auditing. Inspired by NLP-related technologies, the tokenizer parses the source code into a token sequence. One advantage is that the sequence retains strong contextual semantics of the source code. The second is that it cleans up the data noise which was caused by the coder's personal habits such as dependent variable names and indentation. In the meantime, the tokenized source code (TOKEN) can use the language model (LM) for vector space embedding. With the combination of neural network structure and algorithm with time series characteristics such as long short-term memory (LSTM), the model can achieve good performance [17]. Besides, the tokenization method avoids the introduction of additional and more complex data structures. This method is currently relatively reasonable and worth further study. It is also one of the focuses of our article.

The TAP tokenizer [18] proposed by Fang et al. is an effective method to tokenize PHP source code. Based on the self-defined rules, they finished the audit for various vulnerabilities on the CWE dataset. In their tests, the TAP method reached the highest accuracy of 0.9787 and the AUC 0.9941. However, the recall rate of vulnerable code blocks still needs to be improved, which is a common problem in other studies of the same category. The recall rate in experiments described in this paper reached 0.7970 under the proposed TAP algorithm and LSTM model, 0.84 under the bidirectional LSTM model.

In the work of Shigang Liu et al., they presented a system for Cross Domain Software Vulnerability Discovery (CD-VulD) using deep learning and domain adaptation [19]. They converted the cross-domain program representations into token sequences by AST tools, then used token sequences to build a classifier for vulnerability detection. Experimental results show that CD-VulD outperforms the state-of-the-art vulnerability detection approaches by a wide margin.

Their work proved that tokenization is an effective approach and potential research direction in the field of code auditing with machine learning.

In the study [20] of Mukesh et al., they summarized the detailed comparison of indicators in recent researches on source vulnerability auditing using machine learning, as shown in Table 1.

Table 1. Methodology and assessment indicators for other studies.

Authors	Features	Vulnerabilities	Algorithm	Performance
Shar et al.	Static/Dynamic code attributes	SQL, XSS	Logistic regression, MLP	$Recall > 78\%$
Roccardo et al.	Unique-words, Uni_tokens	General	NB, Radom Forest, SVM et al.	$Recall = 82\%$
Mukesh et al.	Proposed_tokens	XSS	Random Forest, SVM, J48 et al	$Recall = 88\%$
Yong Fang et al.	TAP_tokens	XSS, SQL	LSTM	$Recall = 79.7\%$

Due to the universality of XSS threats, using machine learning technology to find XSS vulnerability in source code is a critical research direction. The trigger of XSS often comes from the user input. In XSS-sensitive code blocks, the general code logic is that data from the user input is reviewed or filtered, and finally appears somewhere on the web page [21]. However, there is a problem with both traditional methods and many current machine learning methods: once a filter function used to sanitize the input is detected, the whole code block is considered safe. It ignored the vulnerability caused by an invalid filter. Another extreme situation is that the previously filtering rules set by the coder supposed the use of only one or two cleaning functions is invalid. Then, a large number of filters need to be used to prevent all possible occurrences.

In dynamic Web applications, user input may be output in different locations on HTML pages. That means XSS attackers need to construct specific payloads to accomplish the attack. To intercept attacks from different locations, it is critical to choose a corresponding filtering strategy. An effective filter can greatly reduce the possibility of an XSS attack. However, there will be no help at all if several irrelevant filter functions are used to sanitize the input. In previous studies, researchers focused on source code vectorization and machine learning models themselves. However, they ignored the context information [22] in the source code. This dynamic response and combination reveal that for XSS vulnerability detection in the source code, the context information is more vital, especially the relationship between the filter functions and output locations.

2.2.2. XSS Vulnerability Detection

In the research of Bisht et al., they conclude that filtering is useful as a first level of defense against XSS attacks [23]. However, how to identify the effectiveness of the filter functions remains a challenge. Therefore, they proposed another method detect the content of HTTP response.

Web applications are written implicitly assuming benign inputs and encode programmer intentions to achieve a certain HTML response on these inputs. Maliciously crafted inputs subvert the program into straying away from these intentions, leading to an HTML response that may cause XSS-attacks. In their approach, the main idea for discovering intentions is to generate a shadow response for every (real) HTTP response generated by the web application. The purpose behind generating the shadow response is to elicit the intended set of authorized scripts that correspond to the HTTP response. Whenever an HTTP response is generated by a web application, the model will identify the set of scripts present in the (real) response.

Checking response can effectively prevent malicious script from being inserted into HTML, but the shortcomings of this solution are also obvious. First, the deployment of this scheme must maintain a unique database for each different web application, lacking scalability and flexibility. Besides, this kind of HTTP traffic-based detection will increase the burden of network transmission, and may not work under the influence of network conditions such as DDoS attacks. In contrast, the source code level protection has no impact on the real-time performance of the network, and will not cause

mechanism failure due to bad network conditions [24]. Especially when machine learning technology is combined with real-time processing, problems such as delay, memory, batch processing. will cause greater overhead [25]. The cost of source code audit is so lightweight that almost all websites can apply this program. Thus, it can improve the cybersecurity more generally.

Researchers represented by Rodriguez et al. proposed another XSS detection method [26]. This type of solution determines whether XSS vulnerabilities exist by actively sending various requests (which is very similar to fuzzing) or passively monitoring the HTTP traffic of the target site and analyzing the XSS sensitive payload in the request and response data. This is an effective approach, since the occurrence of XSS and the conditions under which the vulnerability is exploited can be accurately determined based on the real HTTP message content. However, this method may be difficult to perform well in reality. In addition to the above-mentioned real-time-related factors, persistently monitoring and collecting numerous packets will lead to excessive data redundancy. Moreover, for the data samples are likely to be extremely unbalanced, the combination with machine learning will also be hindered.

Besides, fuzzing is currently the most popular vulnerability discovery technique. Conceptually, a fuzzing test starts with generating massive normal and abnormal inputs to target applications, and try to detect exceptions by feeding the generated inputs to the target applications and monitoring the execution states [27,28]. Fuzzing requires little knowledge of targets and could be easily scaled up to large applications, and thus has become the most popular vulnerability discovery solution.

However, considering that fuzzing is a dynamic vulnerability mining method, dynamic program analysis needs to execute the target programs in real systems or emulators. In addition to the construction of the running environment, the analysis of the payload and the runtime environment when the program is abnormal also have high complexity, which indicates fuzzing-related work requires personnel with strong technical skills [29]. This is not friendly to programmers, and coders are more willing to see problems in the source code pointed out directly. Moreover, in this regard, the dynamic nature of fuzzing also leads to code coverage issues is inevitable [30].

In contrast to dynamic analysis, code audit is a representative method of static analysis in vulnerability discovery. Static analysis is the analysis of programs that is performed without actually executing the programs. Unlike fuzzing, static analysis is usually performed on the source code. By analysis on the lexical, grammar, semantics features, data flow analysis, and model checking, static analysis could detect potential vulnerabilities [31]. The advantage of static analysis is the high detection speed. An analyst could quickly check the target code with a static analysis tool and perform the operation timely. However, static analysis endures a high false rate in practice. Due to the lack of easy to use vulnerability detection model, static analysis tools are prone to a large number of false positives. Thus, identifying the results of static analysis remains a tough work.

3. Methodology

Based on the practical significance of the source code vulnerability audit and the comprehensive consideration of the current mainstream research results, our goal is to improve the performance, especially recall rate of the machine learning model for the detection of vulnerable code block samples, that is, to find as many vulnerabilities in source code samples as possible.

3.1. Overview of This Section

This section contains the theoretical underpinnings of the study and an introduction to the various processes. The overall framework is divided into three parts: The first part describes how we process the source code and extract its key information, the second part describes how we introduce the XSS triple into the opcode sequence. The third part describes the theoretical background related to machine learning modeling. Figure 1 shows the framework of our work.

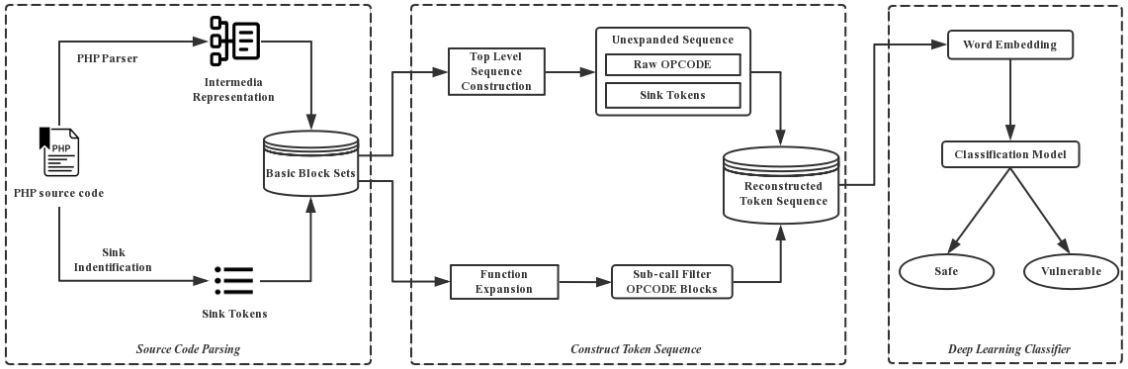

Figure 1. The architecture of the proposed model.

3.2. XSS Triggering in Source Code

The source of XSS triggers often comes from user input. In XSS-sensitive code blocks, the source of data flows is the user inputs, which need to be checked and sanitized [23,32]. Generally, the XSS vulnerability triggering in PHP source code is attributed to inadequate or inappropriate sanitizing (filter) of the user input (source) for a specific output (sink) location [33]. In most cases, "source" is usually an external input from users. What really matters in the source code is the filter–sink corresponding relationship. Evaluations need to be done to judge if the sanitizing of user input is effective, or there could be an XSS vulnerability. However, if an entire project requires auditing, the "source–filter–sink", i.e., the XSS triples, needs to be considered as a whole [34].

Listing 1 shows a typical code block of handing with the input. At line 6, the program takes data from the user using the `$_GET` array. The code that receives user input is defined as "source". At line 9, the program replaces single quotes in user's data by a regular expression. This is because, in many payloads that trigger XSS, quotes are used to close the original content of the document before injection. Functions that sanitize user input like this to prevent XSS triggering are represented by "filter"(or "sanitizer"). At line 11, the data processed by the filter is output in the body tag by the echo function, and the final position written into HTML document is called "sink". At last, the user's input is written to HTML document. As the sink is located in the BODY tag, the attacker can easily insert the script tag in the HTML document with "get" parameter. The way that the single quotation mark is removed does not sanitize it. For example, if the attacker writes payload "`<script>alert(document.cookie)</script>`" in the get parameter, a JavaScript bullet window with the cookie of this website will appear in the web page.

Listing 1: A simple PHP example.

```
1  <!DOCTYPE html>
2  <body>
3  <?php
4      $array = array();
5      $array[] = 'safe';
6      $array[] = $_GET['userData'];
7      $array[] = 'safe';
8
9      $tainted = preg_replace('/\'/', '', $tainted);
10
11     echo $tainted;
12 ?>
13 </body>
14 <html>
```

The source part that receives user input is easy to classify, mostly through PHP's Super Global Variables, as shown in Table 2.

Table 2. Common mode of receiving inputs by PHP.

Super Global Variable	Description
`$_GET`	An associative array of variables passed to the current script via the URL parameters
`$_POST`	An associative array of variables passed to the current script via the HTTP POST method when using application/x-www-form-urlencoded or multipart/form-data as the HTTP Content-Typein the request
`$_COOKIE`	An associative array of variables passed to the current script via HTTP Cookies
`$_REQUEST`	An associative array that by default contains the contents of $_GET, $_POST and $_COOKIE
`$_FILES`	An associative array of items uploaded to the current script via the HTTP POST method
`$_HTTP_RAW_POST_DATA`	A global variable that contains the raw POST data

To extract the "source–filter–sink" from source code without being interfered with by code style, we need to parse the code into another universal form. In our work, a PHP extension called VLD parses PHP source code into internal representation blocks, named OPCODEs.

The output results not only show the operation sequence, but also trace and analyze the data flow, program entry, branch execution, and variable assignment in detail. Unfortunately, VLD does not provide a programming friendly output format. In the use of VLD processing bulk PHP source code, it can not acquire accurate information that we need even with the help of regular matching. In response to it, we perform a secondary development of VLD, adding a programming friendly output format branch. With the aid of the secondarily developed VLD, it is convenient to extract the opcode sequence of a code block.

OPCODE removes some of the unrelated noise from the source code but does not retain the filter-sink content we need. In Figure 2, as the code section of the filter with sink, i.e., the *filter_var* function call with the output position of the echo (in the function setinterval) is simplified for *init_fcall* and echo, missing the key information. Therefore, the subsequent detailed processing for OPCODE is necessary. In this paper, we obtain the function list directly from the supplementary information of the oprands column in the VLD analysis results, which contains the filter function the code block used. However, for complex programs, OPCODE contains more function call information. Moreover, these consistent ECHOs need to be optimized detailed to classify different sinks. Thus, we need to construct a way to effectively recognize the filter functions. The relevant algorithms will be detailed in the following sections. We take one of the OPCODE sequences as an example.

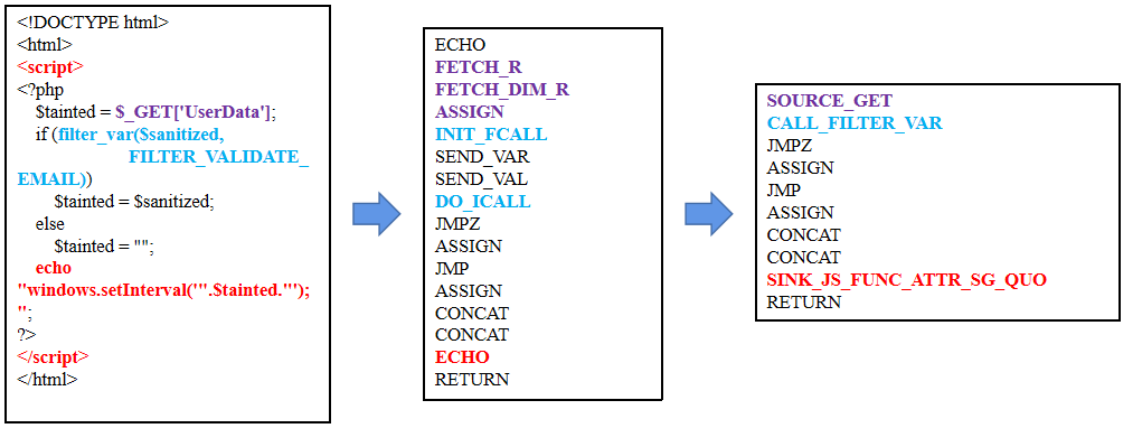

Figure 2. Source versus opcode correspondence.

3.3. OPCODE Sequence Clipping and Optimization

In actual PHP applications, there are usually many function calls. If no distinction is made, connecting the corresponding OPCODE sequences directly according to the function call chain will make the input sequence quite verbose and have a bad influence on the model identification and judgment. To improve the information density of the final input sequence as much as possible, this article uses an OPCODE clipping algorithm for the PHP function call chain.

Filter function: In this paper, the filter function refers to a function that attempts to eliminate potential XSS threats by performing certain processing on untrustworthy input data from users. These functions clean the cross-site script payload that may exist in the input data, effectively preventing the occurrence of XSS. For the XSS payload, from the input point (i.e., source) to the output point (i.e., sink) function cycle, there are only OPCODEs for the data filtering function. The detection model we propose provides effective information.

Filter function recognition algorithm: In the data preprocessing, we can obtain OPCODE sequence information including the complete function call chain with the modified VLD tool. Next, the algorithm determined whether each function in OPCODE belongs to the filtering function, following the whitelist strategy. That is, a function does not contribute to XSS payload cleaning if it does not meet any of the conditions we propose. Here, we list three conditions that related to XSS attacks before the Algorithm 1 [35].

Algorithm 1: isFilter(*block*).

input : *block* is a function block in internal representation
output: *true* or *false*

```
if block contains encoding or escaping then
    return true
else if block contains filtering or replacement then
    return true
else if block contains type conversion then
    return true
else
    return false
end
```

(1) **Encoding / Escaping**: The malicious payloads in most XSS attacks are constructed to add extra tags or attributes to HTML documents, then perform specific operations. In this case, ensuring that payloads from untrusted sources will be output as normal document content is the

primary idea. PHP provides a series of builtin functions for this type of work [36]. By encoding or escaping string data, these functions are intended to make them normal text rather than executable code. For example, the function *htmlentities* converts characters to HTML entities, and the function *addslashes* add backslashes before symbols such as quotation marks and slashes. Besides, in different scenarios, *urlencode*, *htmlspecialchars* can also play this role [37].

(2) **Filtering / Replacement**: Some developers prefer to use some rules set by themselves to filter or replace the XSS payload that may exist in the input data. For example, the function *filter_var* is a filter function commonly used by developers. It can match the specified content through specific expressions. The regular replacement represented by the function *preg_replace* is also a common method to remove XSS vulnerabilities. Such functions are called filtering functions. Analyzing these functions in the code can further audit the effectiveness of the filtering process.

(3) **Type conversion**: In the major of cases, the program expects user input to be specific datatype to ensure its safety. Thus, function like *intval* will be used to compulsorily receive a numeric. However, using this method to force data type conversion of input may drop appropriate input, which is similar to a whitelist strategy.

Depth-first function scanning algorithm: To process a function call chain with multiple layers, this algorithm follows the depth-first strategy and scans every function in the chain recursively. The algorithm is described as follows.

This Algorithm 2 can effectively reduce the function blocks involved in the analysis in the PHP code when processing deep-level function call chains, which increases the information density of the final input sequence. Therefore, we can acquire a simplified and optimized opcode list.

Algorithm 2: scan$(block, seq)$.

```
input  : block is a function block in internal representation
output : seq is the tokens sequence
is_filter ← false;
if isFilter(block) then
    is_filtr ← true
end
foreach line ∈ block do
    if is_filter then
        /* tokenize(line) transforms the line in internal representation to token
           */
        seq ← seq + tokenize(line)
    end
    if line invokes a function call then
        jumps to new function block;
        scan(new_block, seq)
    end
end
```

3.4. Proposed Theory of Sink Identification

The effects of filter–sink source code sequence can be divided into three situations:

(1) User input is output without any filtering. This is the most likely scenario to trigger an XSS vulnerability.

(2) User input is filtered to some extent, but the filtering is insufficient or the filtering function is used incorrectly. Sometimes this approach can prevent part of the XSS attacks, but if the attacker

uses an advanced bypass method, it can bypass the restriction of filter function and trigger XSS vulnerability.

(3) The code fully considers all the filtering conditions of the relevant output and sets excessively strict filtering functions. In this case, it is impossible for an attacker to trigger an XSS vulnerability. However, this kind of situation is very rare in real web applications.

The most ideal way is to use corresponding effective filter functions for various sink types in the code of the program. The research of this paper focuses on the second type, that is, we expect to establish the relationship between various types of filters and sink, and evaluate the defense ability of these pairs against XSS. In Section 3.2, we extract all kinds of filter functions through the scanning algorithm in the rewritten VLD tool. The types of sink are also varied Listing 2 shows several example of sink. However, due to the characteristics of the VLD tool itself, it does not contain the relevant content of the output location. Only an echo type function call symbol as its analysis result. In the analysis of the XSS trigger process, the location of page where the contaminated data will be output is an important factor, so this condition limits our research. In the study by Mukesh et al., the sinks were coarsely classified into 16 types. However, the granularity and accuracy of this manual identification are yet to be evaluated. Thus, we proposed a new filtering algorithm for classifying sinks.

Listing 2: Examples of sinks.

```
<script> echo $tainted; </script>
echo "<".$tainted."href=\"/bob\"/>";
echo "x=\"".$tainted."\"";
echo "body {color:\"".$tainted."\";}";
echo "alert(\""/$tainted."\")";
echo "<div id='".$tainted."'>content</div>";
echo "div onmouseover=\"x=\"".$tainted."\"\>";
```

Based on the classification results of Mukesh et al., and combined with the report of an XSS vulnerability in CVE/CWE [38], we suppose that the identification of sink type needs to be considered based on the following points [39].

The location of output content in the document: Due to the different code logic, untrustworthy input may reach anywhere on the page and be broken through a specifically constructed payload. Based on this, we first classify sink into three categories: (1) HTML, which is the most common type. The user's input will be applied to a certain part of the HTML document. (2) CSS. Some web pages allow the users to change the CSS layout of the page through some options, such as background color, which is often ignored as the XSS vulnerability. (3) JavaScript, in which some untrustworthy parameters will be introduced into the script to participate in the execution. In this case, the code injected by the attacker is easier to break through the original logic and would be more harmful.

HTML tag type of output context: The design of a web page based on the tag makes XSS's payload needs to be carefully constructed to make use of the tag structure of its trigger position. Whether the output position is within the braces of the "style" tag or the content of the "div" tag cannot be generalized. Therefore, the context tag where the output content is located is also an important basis for classification.

Location of output in relation to contextual HTML tags: Based on the second point, the position of the output content in its context needs further analysis. For example, a common example is contaminated data output within a pair of span tags, and sometimes this data will be applied to the "a" tag's "href" attribute. Even in some programs, page tag names or attribute names are designed to be changed by untrustworthy sources. Therefore, these situations deserve a classified discussion.

How the output content is closed: For programmers, what they expect is that the normal user input will be inserted into the corresponding page position. Therefore, the data from the user is

often closed by the original quotes, brackets, and other symbols in the document. For attackers, their constructed payload needs to use the corresponding symbols to close the original symbols in advance, and then insert the part they want to execute. In many cases, preventing attackers at output content closing positions in advance can effectively prevent XSS attacks. For example, the function *addslashes* can add escape characters to quotation marks in data submitted by attackers to convert them into normal text.

Based on four points above, we distinguish the sink type easily. The identification result is shown in Table 3, self-defined OPCODEs are used to replace the single "*ECHO*".

Table 3. Kinds of sinks.

Example	Description	OPCODE
`<script>$output_var</script>`	Output directly in the content part of the script tag	`DATA_SCRIPT`
`<div $output_var=.....>`	Output variable as attribute name of tag	`ATTR_NAME`
`<$output_var attr=.....>`	Output variable as tag name	`TAG_NAME`
`<div id=$output_var>content</div>`	Numeric attribute value	`ATTR_VAL_NO_QUO`
`<div attr='$output_var'>content</div>`	Single quote closed attribute value	`ATTR_VAL_SG_QUO`
`<div attr="$output_var">content</div>`	Double quote closed attribute value	`ATTR_VAL_DB_QUO`
`body{attr:$output_var;}`	Numeric attribute value in CSS block	`CSS_PROP_VAL_NO_QUO`
`body{attr:'$output_var';}`	Single quote closed attribute value in CSS block	`CSS_PROP_VAL_SG_QUO`
`body{attr:"$output_var";}`	Double quote closed attribute value in CSS block	`CSS_PROP_VAL_DB_QUO`
`js_function($output_var);`	Bare parameter in JavaScript function	`JS_FUNC_ATTR_NO_QUO`
`js_function('$output_var');`	Numeric quote closed parameter in JavaScript function	`JS_FUNC_ATTR_SG_QUO`
`js_function("$output_var");`	Double quote closed parameter in JavaScript function	`JS_FUNC_ATTR_DB_QUO`
`<div onclick="$output_var">`	Event handler variable	`EVENT_HANDLER_DB_QUO`
`<span style="$output_var">;`	Inline css style	`CSS_INLINE_STYLE`

3.5. OPCODE Sequence Embedding

Operand sequences are a type of discrete text, and the simple encoding of such text for subsequent classification can cause major difficulties in the gradient optimization process. Therefore, we need suitable methods to map the discrete text into a continuous vector space.

Referring to some methods in the natural language processing field, each opcode sequence can be treated as a document. In this way, an embedding algorithm can transfer the opcode sequence to a vector array.

"Word2vec" [40] is a commonly used pre-training model applied in the field of natural language processing, which can be used to generate a text vector order with contextual properties. It integrates the CBOW, Skip-Gram two language models as well as the MLP neural network structure, which can generate its separate fixed dimension vector for each word in the corpus.

Compared to natural language, the corpus we need to train is the full set of OPCODE tokens. For such a small corpus, the selection of some word2vec hyperparameters requires some fine-tuning. The parameter "window" represents the number of words that are contextually linked. The larger the value, the more context the model will associate. For an opcode sequence that embodies the source code execution process, it includes a whole from input to output, and this data flow is more contextually relevant than natural language. Therefore, we believe that the "window" value should be set large enough that it is reasonable to take a value in the range of 10–20. Besides, we need to reasonably set the dimensional size of each operand token so that it minimizes the dimensionality of the overall operand sequence without losing the amount of information.

3.6. Bi-LSTM Classifier Construction

We implement the Bi-LSTM network to learn the relationships of (source–filter–sink) triples from token sequences.

In the field of natural language processing, the LSTM network [41] is a common neural network structure. However, there is a problem in modeling sequence with LSTM: it cannot encode the information from the back to the front. In our problem, the filter function in the opcode sequence is always in front of the sink. In this situation, LSTM may not be able to acquire the pattern that using the rules of filter function from the sink. Bidirectional LSTM [42] can capture the bidirectional semantic dependence of sequence better by concatenating the network structure of LSTM. The Bi-LSTM structure is shown in Figure 3.

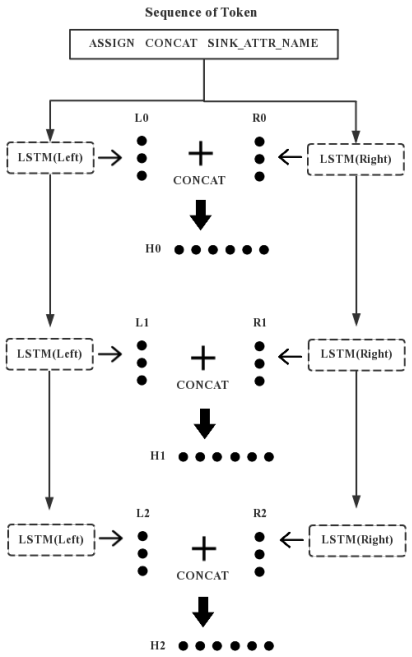

Figure 3. The bidirectional long short-term memory (Bi-LSTM) structure.

In the forward network (LSTM Left), we input "*ASSIGN*", "*CONCAT*", and "*ATTR_NAME*" and get three vectors $\{L_0, L_1, L_2\}$. In the backward network (LSTM right), we input "*ATTR_NAME*", "*CONCAT*", and "*ASSIGN*" and get three vectors $\{R_0, R_1, R_2\}$. Finally, the forward and backward hidden vectors are concatenated and we get $\{[L_0, R_0], [L_1, R_1], [L_2, R_2]\}$ as $\{H_0, H_1, H_2\}$. By doing this, the neural network can learn the context information of the two directions in the whole sequence processing.

For the classification task completed in this paper, we suppose the bidirectional LSTM network is the optimal NN structure since it can deal with all the information of forward and backward.

In our case, due to the different functions of the code, each PHP source code will have different lengths of operation code after conversion to the sequence of operations code. After the embedding process, each word will become a sequence of vectors of unequal length, which does not conform to the input rule of the neural network. We uniformly specify each vector sequence length as L when processing the vector sequence of the sample when L needs to be larger than the maximum sample sequence length. The insufficient parts will be filled with float numeric value 0. The mask layer is used in the front layer of the neural network to mask out the shadow of the 0 values on subsequent calculations. After that, the neural network will access the bidirectional LSTM layer, which contains a two-direction LSTM network structure. The sequence genus of two directions can be learned after concatenating the input sequence. Finally, we append a dense layer as well as a softmax layer to obtain the classification results of the input samples.

Besides, as the MLP network has been used in the word2vec pre-training model for semisupervised learning of the smaller corpus of opcode sequences, we do not import more complex deep neural networks to the classifier for increasing the complexity of the model.

4. Experiments and Performances

4.1. Experiments Setup

In the experiment shown below, we first carried out experiments on different classifiers and compare their performances. Second, for the different degrees of refinement of XSS triples in the OPCODE, we did some comparative experiments on the classifier with the best performance (i.e., the Bi-LSTM classifier) to prove the importance of the triples for PHP source-level XSS vulnerability detection. In the meantime, we gave the curve of loss-accuracy of the validation set and the confusion matrix on the test set throughout the training and evaluating process. At last, we compared the performance of our proposed XSS Guardian with the study done on the same data set to highlight its excellent performance.

Data: We ran experiments on different levels of proposed XSS Guardian. The Common Weakness Enumeration (CWE) is a category system for software weaknesses and vulnerabilities. We adopted CWE-079 as our dataset, which includes 10080 XSS-related PHP source code samples. To be precise, CWE-079 contains 5278 safe code blocks and 4352 unsafe ones. Before these experiments, we have removed possible interference from the code samples, such as extra blank lines and code comments.

Model Training and metric selection: In the neural network training and validating, we use a validation split of 10% in each epoch. As for the traditional machine learning algorithms, we use 10 fold cross-validation for model training and selection of optimal hyperparameters. For the numbers given below, we use the highest experimental metrics on the validation set (no cross with training set).

Hardware: In order to carry out these experiments effectively and precisely, our hardware environment is Intel core i7-9750H, 16 GiB memory, and an Nvidia GTX1660Ti (6GiB).

Evaluation Metrics: Accuracy, F1-Score, Precision, Recall, ROC-AUC, and The Confusion Matrix.

4.2. Comparative Experiment under Different Factors

4.2.1. Comparison of Different TOKEN Sequences

As we mentioned in Section 3, the XSS triples, i.e., SOURCE–FILTER–SINK XSS, can represent the whole I/O data streams of an entire code block. Based on the OPCODE sequence of VLD output, we set up three groups of comparative experiments to measure the impact of triples on XSS vulnerability detection using different combinations of SOURCE, SOURCE+FILTER, and SOURCE+FILTER+SINK. We use two-way LSTM network as a basis to build the classifier. To be more precise, we use the LSTM network with 128 units and make it two-way concatenating. The recurrent dropout probability of this Bi-LSTM layer would be set to 0.25. After this, the network will be connected to three full connection layer activated by the ReLU function to improve its generalization ability. Finally, the upper

layers are connected to a 2-classes-output full connection layer with softmax. The whole network is optimized by the cross-entropy loss function. Such a bi-LSTM classifier structure will also be used in the subsequent experiments.

We trained the classifier to convergence in a group of experiments. The criterion of convergence is that the validation loss of verification set in 10 rounds of training does not decrease any more.

From the experimental results shown in Table 4, in our control group (raw opcode) metrics, the recall rate of vulnerable samples has reached 94.93%, i.e., the sequence structure of opcode has high recognizability for XSS vulnerability identification. However, its recall rate of safe samples only reaches 53.48%, which is close to random guess. This phenomenon indicates that using only the raw opcode sequence without any processing will produce many false positives, that is, normal samples are predicted to be vulnerable samples.

Table 4. Performance of different experiments.

Experiments	Precision		Recall		Macro F1	Mean AUC
	Safe	Vulnerable	Safe	Vulnerable		
OPCODE	0.9402	0.6211	0.5348	0.9493	0.7146	0.7615
OPCODE+SOURCE	0.9408	0.6209	0.5345	0.9495	0.7148	0.7633
OPCODE+SOURCE+FILTER	0.9435	0.6968	0.7062	0.9236	0.7991	0.8483
OPCODE+SOURCE+FILTER+SINK	**0.9813**	**0.8659**	**0.8739**	**0.9785**	**0.9216**	**0.9726**

In the second group, we added source information based on opcode to reflect the way of user input. Yet from the results, there is almost no difference between this group and the control group. Each group of metrics is floating up and down at 0.05%. We conclude that this is due to the subtle difference in neural network convergence points. The comparison shows that the SOURCE is not enough to help distinguish the two samples. This is related to that each sample in our dataset only involves one input and output stream. When the code involves the cross processing of multi-file association and multi-data flow, the determination of the input SOURCE will play a more important role.

In the third group and the fourth group, the addition of FILTER improved the F1 score and AUC by about 8%, and SINK made both metrics break through 90%, and the recall rate of vulnerable samples reached 0.9785. It can be seen that the addition of FILTER and SINK enables the Bi-LSTM-based classifier to learn the characteristics of combination and association of different SOURCE-FILTER-SINK in the source code context, which greatly improves the prediction of XSS vulnerability.

4.2.2. Comparison of Different Classifiers

We also test the performance of the task on different classifiers. In this part, we use the optimization method of SOURCE-FILTER-SINK, that is, to add all the information of XSS triples into the opcode sequence for classification prediction.

Due to the limitations of traditional classifiers on the dimension of input data, we flatten these two-dimensional word vectors and use them as input for classifiers. For random forest classifiers, we use the default parameters in the official Python-sklearn interface. We set an estimator starting value of 4 and a step size of 2 to select the estimator parameter based on the F1 score in the 10-fold cross-validation. The results show that when the number of estimators is greater than 10, the F1 score will reach its convergence and no longer increases. For the support vector classifier, we use the polynomial kernel ($K(x, z) = (\gamma x \cdot z + r)^d$) as the kernel function of the classifier. Compared with radial basis functions, polynomial kernel functions can also produce excellent classification results for nonlinear data distributions without introducing exponential operations of higher powers while reducing time complexity. Plus, we select the "degree" parameter of the polynomial kernel function in cross-validation by setting the degree of 2 and increasing by step of 1. The F1 score converges and reaches its highest value at the degree of 8.

The neural network classifier can directly use the input of two-dimensional word vectors. However, unlike traditional machine learning classifiers, the time complexity of training to convergence for DNN tends to explode as the number of layers increases linearly. Considering this factor, for the convolution network used in the experiment, we set 2 Conv1D layers and 1 maximal pooling layer as a group. By increasing the number of groups gradually and recording the change of F1 score, the F1 score converges when the number of groups equals 3. For classifiers consisting of LSTM networks, we implemented the network structure of the above experiments and recorded the change curves of accuracy and loss. The visualization of experimental data for the hyperparameter selection of each classifier is given by Figures 4 and 5.

(**a**) Hyperparameter Selection of RF "Estimator" (**b**) Hyperparameter Selection of SVM "Degree"

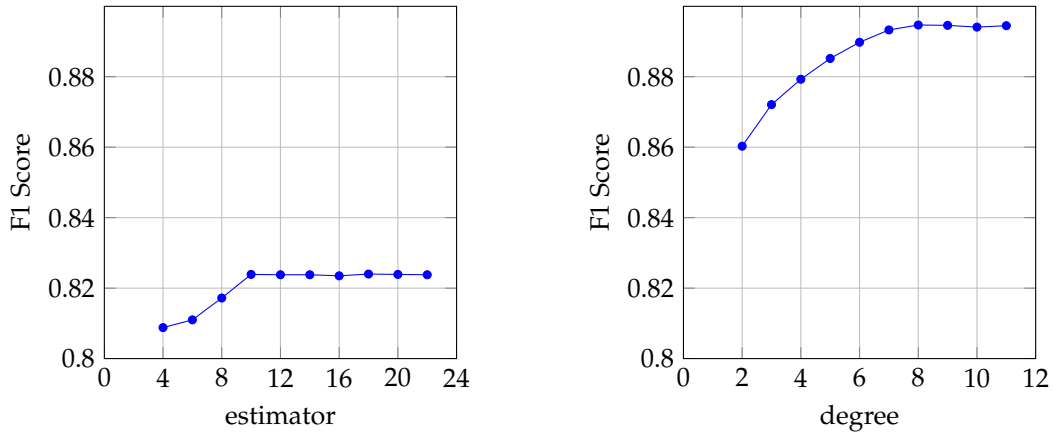

Figure 4. Hyperparameter selection A.

(**a**) Performances on Different CNN Structure (**b**) Performances on Different LSTM Units

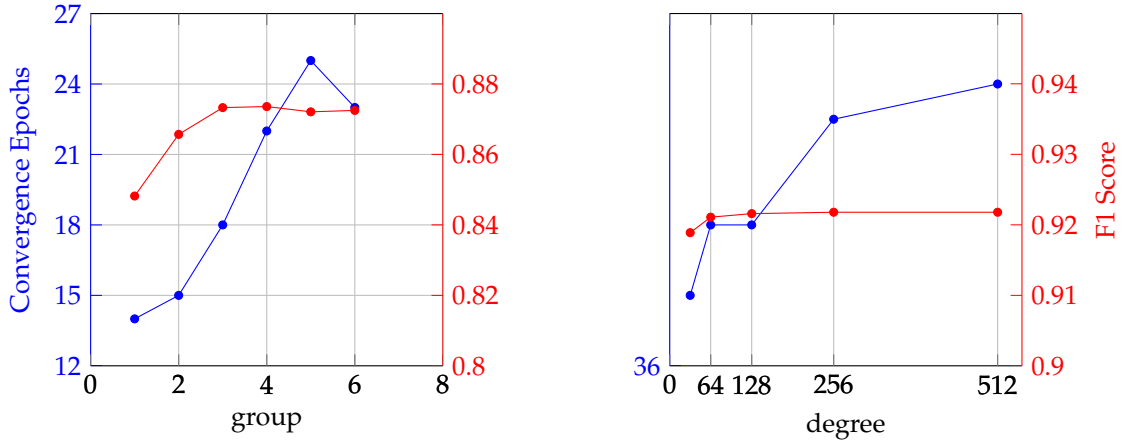

Figure 5. Hyperparameter selection B.

The results show that the classifiers with time series attributes (LSTM and Bi-LSTM) have reached a high metric, showed in Table 5. It only shows the metrics under the best hyperparameter combination. Additionally, the Bi-LSTM classifier has the ability of two-way sequence learning, which can be used to deduce the combination of FILTER and SOURCE from SINK. Thus, it has reached the highest among all metrics. It is worth noting that the SVM-based classifier (SVC) also achieves an excellent score in this experiment. That is closely related to the strong fitting ability of the kernel learning mode of SVC to nonlinear high-dimensional data.

Table 5. Performance of different algorithms.

Algorithm	AUC	F1	Precision	Recall	Vulnerability Recall
Random Forest	0.86	0.82	0.83	0.82	0.90
SVM	0.94	0.89	0.88	0.89	0.94
CNN	0.93	0.87	0.87	0.86	0.87
LSTM	0.95	0.90	0.89	0.90	0.95
Bi-LSTM	**0.97**	**0.92**	**0.92**	**0.92**	**0.98**

On the other hand, we consider that the time consumption of neural networks in the training process is significantly higher than the traditional classifier. For the classifier based on a one-dimensional convolutional neural network, the network will reach its converge point after 20 rounds of training. For the classifier based on the LSTM network, we notice that its convergence time is often longer and more hardware resources are consumed. However, the purpose of the static code audit is to make coders improve the efficiency of vulnerability checks. Unlike the XSS attack load in network traffic, the static audit does not require that high instantaneity. Besides, the neural network classifier based on Bi-LSTM performs significantly better than other classifiers on our evaluations. As a conclusion, we suppose such a time consumption is reasonable and acceptable. Besides, the code audit is generally performed after the program source code is finished, which will not affect the application in the production environment. Therefore, the model does not require real-time performance at the network transmission level.

4.3. Generalization Capability Assessment

4.3.1. Evaluation on the Test Dataset

We selected data samples (approximately 20%) from the dataset that did not cross training sets or validation sets for testing. We use the proposed XSS Guardian classifier, that is, Bi-LSTM classifier with the XSS triples optimization to run the test. The results are given in the form of a standardized confusion matrix in Figure 6. For these safe samples, XSS Guardian still has an FPR of 11.76%, yet relatively small proportions. In the meantime, for the XSS-vulnerable samples, XSS Guardian has minimized the number of FNR to 2.83%.

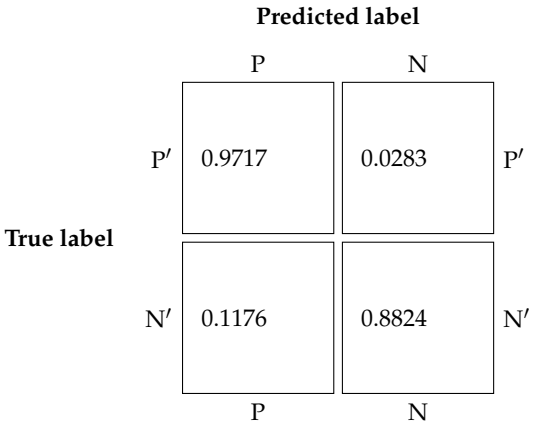

Figure 6. Standardized Confusion Matrix.

4.3.2. Comparison with Similar Studies

We believe that the TOKEN method proposed by Mukesh et al. is quite effective for the vulnerability detection of XSS. However, they did not give the published experimental code resources

in the paper, only descriptive pseudocode for reference. Therefore, it is a pity for us not to reproduce their work here. However, the data set used in their paper was consistent with ours. The only enhancement we made on the dataset is to clean the spelling errors and useless code comments. Thus, we briefly cite the experimental data of this article here and make a rough comparison with the metrics of XSS Guardian in Table 6.

We found that all metrics of XSS Guardian had a certain degree of increase. Among these the recall rate had increased by 11.23%, and the F1-Score had also increased by 7.29%, due to the more detailed TOKEN sequence generation algorithms and the more advanced deep learning classifiers. The accuracy and loss curve of the XSS Guardian (Bi-LSTM model) is shown in Figure 7.

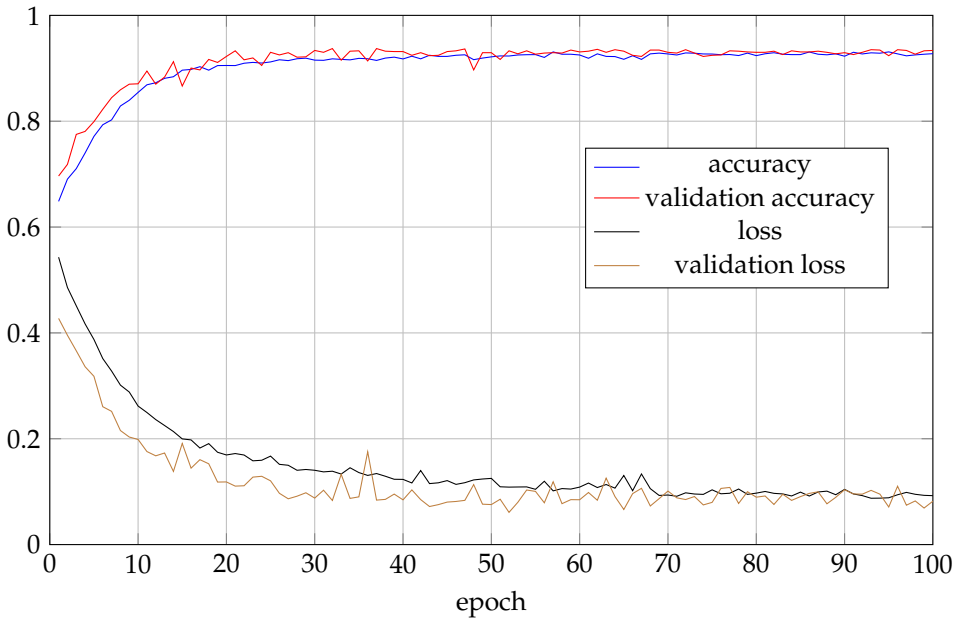

Figure 7. Accuracy and loss curve.

Table 6. Performance of different methods.

Method	Precision	Recall	F1	Accuracy
Mukesh's method	0.8870	0.8300	0.8590	0.8890
XSS Guardian	0.9200	0.9232	0.9216	0.9237
Improvements	**+3.72%**	**+11.23%**	**+7.29%**	**+2.36%**

5. Conclusions and Future Work

In this paper, we proposed the "Cross-Site Scripting Guardian"—a novel approach to detect XSS vulnerability in PHP source code based on machine learning. Using our rewritten VLD, we can extract the detailed operation code and the XSS triples that describe how the code dealing with the user input. Thus, our method can be easily implemented and further extended by researchers in the follow-up research. In the experiment on the test dataset, the recall rate for vulnerability samples is as high as 98%, which also shows the importance of learning the opcode sequence and the pattern of input–output in the data stream.

Besides, compared with traditional static analysis tools, the way of using a machine learning model to build a code audit detector would not be affected by the strong dependence of detection tools on matching rules. Users can constantly enrich the training samples to further improve the detection ability of the model.

In future work, two directions might be further explored: First, the impact of our method on more complex data flow sources is limited. However, some hard-to-find vulnerabilities have multiple

execution branches and multiple file associations. Thus, the analysis of the multiple execution branch source code vulnerability analysis is a meaningful research direction. Second, we also try to extend our method to other web application vulnerability audits, such as SQL injection vulnerability mining. However, to improve accuracy and efficiency, we need to propose an approach that is highly related to the characteristics of that vulnerability. Only in this way, machine learning technology can be further widely used and more in-depth in the field of code audit.

Author Contributions: Conceptualization, C.H., C.L., and Y.W.; methodology, C.H., C.L., and Y.W.; validation, C.L., Y.W., and C.M.; formal analysis, Y.W.; investigation, C.L.; data curation, C.L., Y.W., and C.M.; writing—original draft preparation, C.L. and Y.W; writing—review and editing, C.L., Y.W., C.M., and C.H.; supervision, C.H.; project administration, C.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by the National Natural Science Foundation of China (No.61902265), the Key Research and Development Plan Project of Sichuan Province (No.2020YFG0047), Sichuan University Postdoc Research Foundation (No.2019SCU12068), and Guangxi Key Laboratory of Cryptography and Information Security (No.GCIS201921).

Acknowledgments: This work was completed under the guidance of Cheng Huang of Sichuan University, China.

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

References

1. Nirmal, K.; Janet, B.; Kumar, R. It's More Than Stealing Cookies-Exploitability of XSS. In Proceedings of the 2018 Second International Conference on Intelligent Computing and Control Systems (ICICCS), Madurai, India, 14–15 June 2018; pp. 490–493.
2. Web Applications Vulnerabilities and Threats: Statistics for 2019. Available online: https://www.ptsecurity.com/ww-en/analytics/web-vulnerabilities-2020/ (accessed on 20 February 2020).
3. Chen, H.; Huang, B.; Chen, W.; Liu, F . *Machine Learning Principles and Applications*; University of Electronic Science and Technology Press: Chengdu, China, 2017.
4. Chen, X.; Li, M.; Jiang, Y.; Sun, Y. A Comparison of Machine Learning Algorithms for Detecting XSS Attacks. In Proceedings of the International Conference on Artificial Intelligence and Security, New York, NY, USA, 26–28 July 2019; pp. 214–224.
5. Miao, C. VLD with JSON Format Output Support. Available online: https://github.com/ChanthMiao/vld (accessed on 20 February 2020).
6. Top 10 Web Application Security Risks. Available online: https://owasp.org/www-project-top-ten/ (accessed on 20 February 2020).
7. Aghaei, S.; Nematbakhsh, M.A.; Farsani, H.K. Evolution of the world wide web: From WEB 1.0 TO WEB 4.0. *Int. J. Web Semant. Technol.* **2012**, *3*, 1–10.
8. Gupta, K.; Singh, R.R.; Dixit, M. Cross site scripting (XSS) attack detection using intrustion detection system. In Proceedings of the 2017 International Conference on Intelligent Computing and Control Systems (ICICCS), Madurai, India, 15–16 June 2017; pp. 199–203.
9. Yamaguchi, F.; Lindner, F.; Rieck, K. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In Proceedings of the 5th USENIX Conference on Offensive Technologies, San Francisco, CA, USA, 8 August 2011; p. 13.
10. Lingzi, X.; Zhi, L. An Overview of Source Code Audit. In Proceedings of the 2015 International Conference on Industrial Informatics-Computing Technology, Intelligent Technology, Industrial Information Integration, Wuhan, China, 3–4 December 2015; pp. 26–29.
11. Choi, Y.H.; Liu, P.; Shang, Z.; Wang, H.; Wang, Z.; Zhang, L.; Zhou, J.; Zou, Q. Using Deep Learning to Solve Computer Security Challenges: A Survey. *arXiv* **2019**, arXiv:1912.05721.
12. Liu, S.; Lin, G.; Han, Q.; Wen, S.; Zhang, J.; Xiang, Y. DeepBalance: Deep-Learning and Fuzzy Oversampling for Vulnerability Detection. *IEEE Trans. Fuzzy Syst.* **2019**, *28*, 1329–1343.
13. Alon, U.; Zilberstein, M.; Levy, O.; Yahav, E. code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.* **2019**, *3*, 1–29.
14. Dahse, J.; Schwenk, J. RIPS-A static source code analyser for vulnerabilities in PHP scripts. In *Seminar Work (Seminer Çalismasi)*; Horst Görtz Institute Ruhr-University Bochum: Bochum, Germany, 2010.

15. Source Code Security Audit. Available online: https://github.com/WhaleShark-Team/cobra (accessed on 20 February 2020).
16. WordPress Version 1.5. Available online: https://github.com/WordPress/WordPress/tree/1.5-branch(accessed on 20 February 2020).
17. Guo, N.; Li, X.; Yin, H.; Gao, Y. VulHunter: An Automated Vulnerability Detection System Based on Deep Learning and Bytecode. In Proceedings of the International Conference on Information and Communications Security, Beijing, China, 15–17 December 2019; Springer: Berlin/Heidelberg, Germany, 2019; pp. 199–218.
18. Fang, Y.; Han, S.; Huang, C.; Wu, R. TAP: A static analysis model for PHP vulnerabilities based on token and deep learning technology. *PLoS ONE* **2019**, *14*, e0225196.
19. Liu, S.; Lin, G.; Qu, L.; Zhang, J.; De Vel, O.; Montague, P.; Xiang, Y. CD-VulD: Cross-Domain Vulnerability Discovery based on Deep Domain Adaptation. *IEEE Trans. Dependable Secur. Comput.* **2020**, 1, doi:10.1109/TDSC.2020.2984505.
20. Gupta, M.K.; Govil, M.C.; Singh, G. Predicting Cross-Site Scripting (XSS) security vulnerabilities in web applications. In Proceedings of the 2015 12th International Joint Conference on Computer Science and Software Engineering (JCSSE), Songkhla, Thailand, 22–24 July 2015; pp. 162–167.
21. Yan, F.; Qiao, T. Study on the Detection of Cross-Site Scripting Vulnerabilities Based on Reverse Code Audit. In Proceedings of the International Conference on Intelligent Data Engineering and Automated Learning, Yangzhou, China, 12–14 October 2016; Springer: Berlin/Heidelberg, Germany, 2016; pp. 154–163.
22. Steffens, M.; Rossow, C.; Johns, M.; Stock, B. Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild. In Proceedings of the NDSS Symposium, San Diego, CA, USA, 24–27 February 2019.
23. Bisht, P.; Venkatakrishnan, V. XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks. In Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Paris, France, 10–11 July 2008; Springer: Berlin/Heidelberg, Germany, 2008; pp. 23–43.
24. Di Mauro, M.; Di Sarno, C. A framework for Internet data real-time processing: A machine-learning approach. In Proceedings of the 2014 International Carnahan Conference on Security Technology (ICCST), Rome, Italy, 13–16 October 2014; pp. 1–6.
25. Batyuk, A.; Voityshyn, V. Apache storm based on topology for real-time processing of streaming data from social networks. In Proceedings of the 2016 IEEE First International Conference on Data Stream Mining & Processing (DSMP), Lviv, Ukraine, 23–27 August 2016; pp. 345–349.
26. Rodriguez, G.; Torres, J.; Flores, P.; Benavides, E.; Nuñez-Agurto, D. XSStudent: Proposal to Avoid Cross-Site Scripting (XSS) Attacks in Universities. In Proceedings of the 2019 3rd Cyber Security in Networking Conference (CSNet), Quito, Ecuador, 23–25 October 2019; pp. 142–149.
27. Manès, V.J.M.; Han, H.; Han, C.; Cha, S.K.; Egele, M.; Schwartz, E.J.; Woo, M. The art, science, and engineering of fuzzing: A survey. *IEEE Trans. Softw. Eng.* **2019**, doi:10.1109/TSE.2019.2946563.
28. Falana, O.J.; Ebo, I.O.; Tinubu, C.O.; Adejimi, O.A.; Ntuk, A. Detection of Cross-Site Scripting Attacks using Dynamic Analysis and Fuzzy Inference System. In Proceedings of the 2020 International Conference in Mathematics, Computer Engineering and Computer Science (ICMCECS), Ayobo, Ipaja, Lagos, Nigeria, 18–21 March 2020; pp. 1–6.
29. Li, Y.; Ji, S.; Lv, C.; Chen, Y.; Chen, J.; Gu, Q.; Wu, C. V-fuzz: Vulnerability-oriented evolutionary fuzzing. *arXiv* **2019**, arXiv:1901.01142.
30. Marashdih, A.W.; Zaaba, Z.F.; Suwais, K.; Mohd, N.A. Web Application Security: An Investigation on Static Analysis with other Algorithms to Detect Cross Site Scripting. *Procedia Comput. Sci.* **2019**, *161*, 1173–1181.
31. Li, J.; Zhao, B.; Zhang, C. Fuzzing: A survey. *Cybersecurity* **2018**, *1*, 6.
32. Yusof, I.; Pathan, A.S.K. Preventing persistent Cross-Site Scripting (XSS) attack by applying pattern filtering approach. In Proceedings of the The 5th International Conference on Information and Communication Technology for The Muslim World (ICT4M), Kuching, Malaysia, 17–18 November 2014; pp. 1–6.
33. Mohammadi, M.; Chu, B.; Lipford, H.R. Automated Repair of Cross-Site Scripting Vulnerabilities through Unit Testing. In Proceedings of the 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Berlin, Germany, 27–30 October 2019; pp. 370–377.
34. Yan, X.X.; Wang, Q.X.; Ma, H.T. Path sensitive static analysis of taint-style vulnerabilities in PHP code. In Proceedings of the 2017 IEEE 17th International Conference on Communication Technology (ICCT), Chengdu, China, 27–30 October 2017; pp. 1382–1386.

35. Elkhodr, M.; Patel, J.K.; Mahdavi, M.; Gide, E. Prevention of Cross-Site Scripting Attacks in Web Applications. In Proceedings of the Workshops of the International Conference on Advanced Information Networking and Applications, Caserta, Italy, 15–17 April 2020; Springer: Berlin/Heidelberg, Germany, 2020; pp. 1077–1086.
36. Zubarev, D.; Skarga-Bandurova, I. Cross-Site Scripting for Graphic Data: Vulnerabilities and Prevention. In Proceedings of the 2019 10th International Conference on Dependable Systems, Services and Technologies (DESSERT), Leeds, UK, 5–7 June 2019; pp. 154–160.
37. Papagiannis, I.; Migliavacca, M.; Pietzuch, P. PHP Aspis: Using partial taint tracking to protect against injection attacks. In Proceedings of the 2nd USENIX Conference on Web Application Development, Portland, OR, USA, 15–16 June 2011; Volume 13.
38. CWE-79: Improper Neutralization of Input During Web Page Generation (Cross-site Scripting). Available online: https://cwe.mitre.org/data/definitions/79.html (accessed on 20 February 2020).
39. Gupta, S.; Gupta, B. A robust server-side javascript feature injection-based design for JSP web applications against XSS vulnerabilities. In *Cyber Security*; Springer: Berlin/Heidelberg, Germany, 2018; pp. 459–465.
40. Mikolov, T.; Chen, K.; Corrado, G.; Dean, J. Efficient estimation of word representations in vector space. *arXiv* **2013**, arXiv:1301.3781.
41. Calzavara, S.; Conti, M.; Focardi, R.; Rabitti, A.; Tolomei, G. Machine Learning for Web Vulnerability Detection: The Case of Cross-Site Request Forgery. *IEEE Secur. Priv.* **2020**, *18*, 8–16.
42. Graves, A.; Schmidhuber, J. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Netw.* **2005**, *18*, 602–610.