CVE VULNERABILITY CLASSIFICATION IN SOURCE CODE BASED ON TOKEN ANALYSIS AND LSTM NETWORKS

Van Cong Nguyen[1], *Huy Toan Le*[2], *Minh Thanh Ta*[1,*]

Abstract

As web applications become increasingly widespread, the importance of source code security is growing rapidly. Exposed vulnerabilities present serious risks to both service providers and customers. Various models have been proposed to address this issue, however, most approaches rely on complex graph structures generated from source code or on expert-driven regular expression patterns. This paper introduces a model that utilizes token-based mechanisms combined with deep learning techniques for efficient vulnerability detection in PHP (Hypertext Preprocessor) web applications. By leveraging the PHP tokenization process, we have developed a custom token that merges tokens, supports key PHP features, and optimizes parsing. Using datasets such as the Software Assurance Reference Dataset (SARD) and SQL Injection Labs (SQLI-LABS), this paper demonstrates the training of a deep learning model with enhanced tokens to effectively detect vulnerabilities in the source code.

Index terms

Tokens; source code; deep learning; vulnerability detection; natural language processing; PHP source code.

1. Introduction

The Internet plays a critical role in politics, economics, culture, and social life. With the emergence of various Internet environments, such as web applications [1], cloud computing [2], and community service platforms [3], numerous security challenges have arisen. As open source web applications evolve rapidly, so do cyber threats. Research by F5 Labs indicates that web applications are the source of 53% malicious attacks, based on an analysis of 433 significant incidents over 12 years. Thus, securing web applications are essential for effectively combating cyber intrusions.

[1]Institute of Information and Communication Technology, Le Quy Don Technical University

[2]Department of Digital Transformation and Environment Resources Data Information – Ministry of Natural Resources and Environment.

[*]Corresponding author, email: thanhtm@lqdtu.edu.vn

DOI: 10.56651/lqdtu.jst.v13.n02.928.ict

Among the top 10 million websites listed on Alexa, 55.2% are built on content management systems. In particular, WordPress, Joomla! and Drupal dominate the market, collectively accounting for 69% of its share, and all of these systems are developed using PHP.

As the programming language that powers millions of websites, the security issues associated with PHP code cannot be overlooked. Traditionally, the security of PHP applications was ensured through manual testing, a method that proved to be complex, time-consuming, and labor-intensive. Moreover, manual testing is ill-suited for addressing the needs of extensive modern open-source codebases. Along with the development of source code static analysis technology [4] currently, many rely on regular expression patterns that leverage vulnerability detection features derived from expert knowledge. Prominent tools such as RIPS [5] and Pixy [6] offer faster alternatives to traditional manual testing. However, these tools require specific features to effectively identify vulnerabilities, which can be impractical for complex vulnerabilities. The rise of machine learning has prompted an increasing number of researchers to explore innovative approaches in both machine learning and deep learning, often involving the manual extraction of features for traditional models or the identification of sensitive obfuscation functions to support vulnerability detection.

1.1. Challenges

Detecting security vulnerabilities through source code analysis requires a deep understanding of both the security principles and the structural semantics of the code. The inherent complexity of the source code poses significant challenges in applying deep learning techniques effectively.

In extensive and long-established systems, source code often contains intricate structures, such as nested conditional statements and loops, complicating the analysis and understanding of the code's context. In addition, tracking the usage of variables across different segments adds to this complexity.

Although source code is typically represented as text, converting it into numerical formats (e.g., vectors) are essential for deep learning applications. Effectively capturing the complex structure of the code in a form comprehensible to deep learning models presents a significant challenge, particularly in preserving the syntax and semantics of the original code. Security vulnerabilities may not be immediately discernible from the text alone; they often become apparent only when the code is analyzed within its complete context.

The complexity of the source code introduces considerable hurdles in the application of deep learning for vulnerability detection. Navigating complex code structures, dynamic contexts, and interactions with other system components requires the development and application of advanced techniques to ensure that deep learning models perform effectively. Addressing these challenges will enhance vulnerability detection capabilities and contribute to broader advancement of software security.

1.2. Contributions

This paper introduces a novel approach for processing PHP source code data using token technology and deep learning to identify security vulnerabilities. The primary contributions of this work are summarized as follows:

1) Methods for transforming data tokens to optimize the analysis of data flow are investigated and experimented with. The source code is abstractly converted into tokens, which are then analyzed for vulnerability detection in PHP applications using deep learning technologies.
2) By enhancing token technology in conjunction with deep learning, we achieve improved learning speeds and accuracy compared to previously tested models. Experimental evaluations demonstrate that our proposed model attains an accuracy of 0.9558 in label classification and reaches 0.9987 on the CWE-89 and CWE-90 datasets, while reducing training times compared to older models using the same SARD datasets.

1.3. Structure

The paper is structured as follows: Section 2 reviews relevant literature on the evaluation of CVE security vulnerabilities in source code. In Section 3-A, we introduce our proposed approach, which enhances the token extraction process and the generation of feature vectors for LSTM training. Section 4 details the experimental methodology, including the dataset, experimental setup, implementation process, and results. Finally, Section 5 concludes the paper and discusses directions for future research.

2. Related works

In static source code analysis, two main approaches are used for detecting security vulnerabilities: tools based on expert knowledge and machine learning technologies. Traditional tools like Pixy and RIPS specialize in detecting XSS and analyzing PHP code, respectively. SAFERPHP [7] combines taint analysis with Control Flow Graph (CFG) [8] to detect semantic vulnerabilities. However, these tools require continuous updates to keep pace with emerging attack techniques and evolving codebases [9]- [12].

Recently, machine learning methods have significantly advanced vulnerability detection. VulDeePecker employs LSTM neural networks to analyze source code [13], [14], and NAVEX [15] integrates dynamic execution [16]. However, many tools still have limitations in detecting diverse vulnerabilities and depend on fixed sink points, leading to high costs and frequent updates. A previous model using token transformation achieved an accuracy of 0.9357 [17]- [20]. To enhance its performance, we have developed an improved token transformation method.

3. Proposed solution

3.1. Proposed solution

This paper applies a model for analyzing PHP source code to identify and detect CVE vulnerabilities. Based on the characteristics of the PHP language, the tokens are specially customized to analyze strings, numbers, and functions, based on the original token mechanism. These tokens are then merged, and the variables are repeated to achieve data flow analysis, aiming to preserve only the lines of code that contain processing functions as illustrated in the flow in Figure 1.

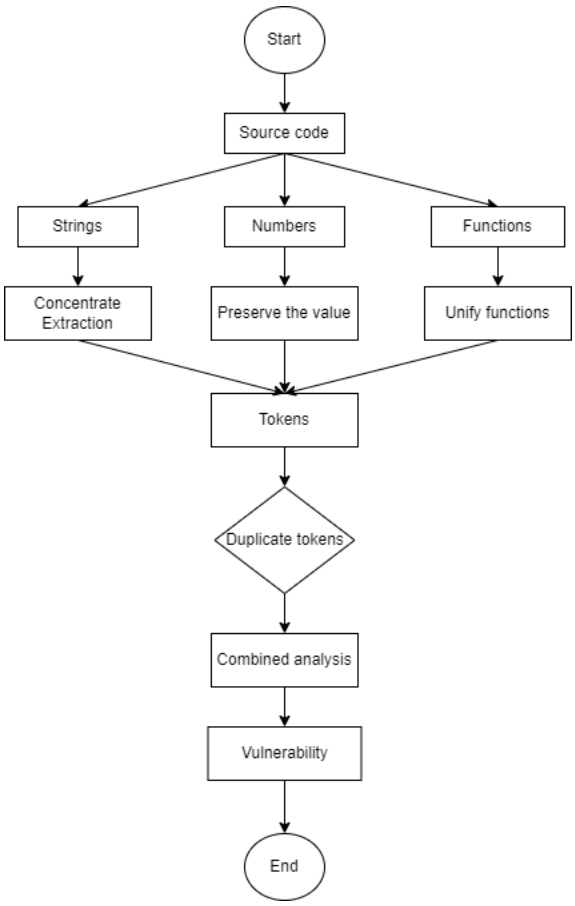


Fig. 1. Flowchart of the proposed solution.

The result of this process is the use of the "word2vec" model to convert tokens into vectors, followed by the application of an LSTM network to train the model to detect over 7 different types of vulnerabilities As illustrated in the overview diagram in Figure 2. The `token_get_all()`[1] function in PHP can analyze code into PHP tokens using the Zend tool. Each keyword and symbol in PHP is translated into a

[1]https://www.php.net/manual/en/function.token-get-all.php

token that begins with an uppercase letter T. This function can also recognize HTML code, comments, and composite symbols, while also locating the position of the line of code where they appear. However, one limitation of this function is that it analyzes any string into the same token `T_CONSTANT_ENCAPSED_STRING`, leading to the loss of detailed information about the content of the string, and the same occurs with numbers.

3.2. String analysis

Moreover, the `token_get_all()` function only recognizes PHP keywords and a limited number of functions. Other identifiers, such as class names, are treated as the token `T_STRING`, making it difficult for the deep learning model to analyze PHP code. Therefore, it is necessary to optimize the standard PHP token parser.

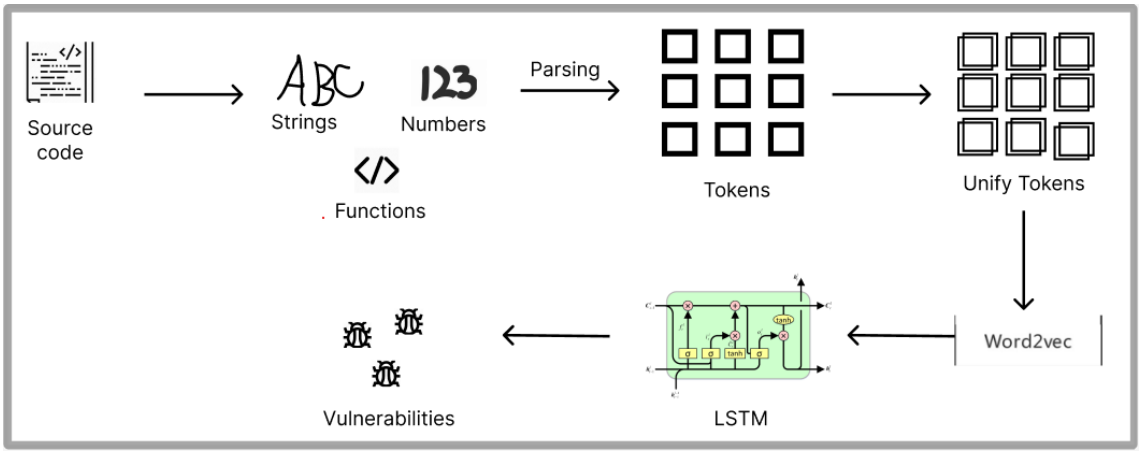


Fig. 2. Overview diagram.

Based on previous knowledge of security vulnerabilities, we believe that the boundary symbols of strings are more likely related to SQL injection and other vulnerabilities than the content of the entire string. Thus, the custom token parser will scan each string to extract boundary symbols. For example, a constant string `SELECT ... FROM A where A.B ='` will only be parsed as `T_CONSTANT_ENCAPSED_STRING` by the built-in `token_get_all()` function, but the token parser will analyze it as `T_CONSTANT_ENCAPSED_STRING'` to highlight the single quote. Coupled with the complexity of the source code, this will always lead to duplication in combining token codes, which in turn combines the codes. For example, a string `T_CONSTANT_ENCAPSED_STRING '` combined with another `T_CONSTANT_ENCAPSED_STRING '` while ensuring separate symbols means we only combine them when the codes have matching distinct symbols like `2_T_CONSTANT_ENCAPSED_STRING '`.

Additionally, there is a specific case in PHP where variables will be parsed when they are within double quotes. Therefore, boundary symbols not only refer to the first and last characters of the string but also to the characters adjacent to the variables located within the double quotes.

```
Input:

<?php
$sql1 = " select * from admin where id=' $_GET[x] ;";
$sql2 = " select * from admin where id="" . $_GET[x] . ";";
?>

Output of inbuilt function:

//$sql1
"T_ENCAPSED_AND_WHITESPACE T_VARIABLE [T_STRING ]T_ENCAPSED_AND_WHITESPACE ";
//$sql2
T_CONSTANT_ENCAPSED_STRING .T_VARIABLE [T_STRING ].T_CONSTANT_ENCAPSED_STRING ;

Output of our tokenizer:
//$sql1
T_CONSTANT_ENCAPSED_STRING ' T_INPUT [ x ] T_CONSTANT_ENCAPSED_STRING ;
//$sql2
T_CONSTANT_ENCAPSED_STRING ' . T_INPUT [ x ] . T_CONSTANT_ENCAPSED_STRING ;
```

Fig. 3. Comparison of analysis in situations with different boundary symbols.

As shown in Figure 3, `$sql1` and `$sql2` are different expressions of the same SQL query. The built-in PHP function `token_get_all()` parses them into different token strings. Specifically, `$sql1` has additional double quotes separating the string, which is meaningless. This inconsistency may lead the model to misinterpret the code.

The new token parser can accurately recognize various boundary situations and special boundary characters. Additionally, our token parser unifies the tokens in multiple forms for easier identification.

3.3. Parsing numbers

Integer types often play a crucial role in configuration options, and many configuration errors can arise from the use of incorrect integer values. However, if all integers are assigned a common token `T_LNUMBER`, the model will be unable to distinguish these values in their specific contexts, leading to an inability to accurately identify configuration options.

To address this issue, integers are classified based on their usage context rather than being converted into a common token. This approach allows the model to accurately recognize and process integer values within their specific contexts, ensuring precise analysis and detection of numerical values. Additionally, it helps maintain the integrity of integer values in configuration options, preventing the loss of crucial information when handling integers in different scenarios.

This helps the model effectively detect configuration-related issues, thereby improving its ability to recognize and handle complex configuration scenarios.

3.4. Parsing functions

The `token_get_all()` function in PHP is a useful tool for parsing PHP source code into tokens. However, one limitation of this function is that it cannot distinguish between different function names; all function names are parsed into the token `T_STRING`. To address this issue, we use the `function_exists()` function to check whether a function is actually defined. Based on the result of `function_exists()`, we retain the function name if it is identified as a defined function or if its tokens are `T_STRING`. This helps preserve the values of functions in the source code without being completely transformed into a common token in the original code.

3.5. Token unification

In source code analysis, different functions may have different names but serve the same purpose, making it harder for deep learning models to classify them correctly. In PHP, many functions perform similar operations, such as including a local file, despite having different names. To reduce confusion, we maintain a list that unifies tokens of similar functions. This helps group functions with the same functionality under a common token, reducing complexity and improving accuracy (Table 1).

For example, in the fifth row of Table 1, functions like `include()`, `include_once()`, `require()`, and `require_once()` are analyzed into different tokens by PHP. However, they all serve the same purpose: including a local file. Our token parser unifies them into a common token: `T_INCLUDES`, allowing the model to recognize their shared functionality. Since relying solely on PHP's built-in token analysis is insufficient, we also incorporate function names to ensure consistency.

Additionally, a vulnerable line of code typically contains at least one function, whereas statements like assignments or variable declarations have little direct relevance to security vulnerabilities. Based on this principle, our token parser retains only lines containing functions while replacing repeated variables with their corresponding names. This approach optimizes the learning process, particularly in handling parameter passing and detecting security-relevant code lines.

Through retaining only the lines of code that execute functions, the token parser helps the model focus on the most critical parts of the code. Additionally, accurately analyzing the content of the function parameters, such as identifying elements in an array, helps preserve the semantics of the source code. The relationships between words and their order are carefully processed, ensuring that the positional change of a word does not alter the overall meaning. Furthermore, user-defined functions are also analyzed in detail, allowing for the creation of corresponding tokens, which facilitates more effective analysis and vulnerability detection.

Table 1. Unified tokens

Unified Tokens	Token Names or Function Names
T_IGNORE	T_DOC_COMMENT, T_COMMENT, T_INLINE_HTML, T_WHITESPACE, T_OPEN_TAG, T_CLOSE_TAG
T_ASSIGNMENT	T_AND_EQUAL, T_CONCAT_EQUAL, T_DIV_EQUAL, T_MINUS_EQUAL, T_MOD_EQUAL, T_MUL_EQUAL, T_OR_EQUAL, T_PLUS_EQUAL, T_SL_EQUAL, T_SR_EQUAL, T_XOR_EQUAL
T_COMPARISON	T_IS_EQUAL, T_IS_GREATER_OR_EQUAL, T_IS_IDENTICAL, T_IS_NOT_EQUAL, T_IS_NOT_IDENTICAL, T_IS_SMALLER_OR_EQUAL
T_INCLUDES	T_INCLUDE, T_INCLUDE_ONCE, T_REQUIRE, T_REQUIRE_ONCE
T_ECHO	T_PRINT, T_ECHO, T_EXIT, T_OPEN_TAG_WITH_ECHO, print_r, printf, vprintf, trigger_error, user_error, odbc_result_all, ifx_htmltbl_result
T_INPUT	`$_GET`, `$_POST`, `$_COOKIE`, `$_REQUEST`, `$_FILES`, `$_SERVER`, `$HTTP_GET_VARS`, `$HTTP_POST_VARS`, `$HTTP_COOKIE_VARS`, `$HTTP_REQUEST_VARS`, `$HTTP_POST_FILES`, `$HTTP_SERVER_VARS`, `$HTTP_RAW_POST_DATA`, `$argc`, `$argv`, get_headers, getallheaders, get_browser, import_request_variables
T_PREG	preg_filter, preg_grep, preg_last_error, preg_match_all, preg_match, preg_quote, eregi, preg_replace_callback, preg_replace, ereg_replace, ereg, eregi_replace
T_EXEC	backticks, exec, expect_popen, passthru, pcntl_exec, popen, eval, proc_open, shell_exec, system
T_SQL	dba_insert, dba_fetch, dba_delete, dbx_query, odbc_do, odbc_exec, odbc_execute, db2_exec, db2_execute, fbsql_db_query, fbsql_query, ibase_query, ibase_execute, ifx_query, ifx_do, ingres_query, ingres_execute, ingres_unbuffered_query, msql_db_query, msql_query, mssql_query, sybase_unbuffered_query, mssql_execute, mysql_db_query, mysql_query, mysql_unbuffered_query, mysqli_stmt_execute, mysqli_query, mysqli_real_query, mysqli_master_query, oci_execute, ociexecute, ovrimos_exec, ovrimos_execute, ora_do, ora_exec, pg_query, pg_send_query, pg_send_query_params, pg_send_prepare, pg_prepare, sqlite_open, sqlite_popen, sqlite_array_query, arrayQuery, singleQuery, sqlite_query, sqlite_exec, sqlite_single_query, sqlite_unbuffered_query, sybase_query

3.6. Processing duplicate tokens

In our experiments, we found that multiple consecutive identical token codes caused difficulties in processing transformations with many duplicates, reducing accuracy. To retain essential components while improving token-to-vector conversion with the word2vec model, we merged consecutive duplicate tokens into one and prefixed the repetition count before the token code.

Initially, we merged all consecutive token codes, regardless of separators like periods or commas. However, this did not improve accuracy and caused confusion between token labels. Re-evaluating the approach, we found it necessary to preserve boundary symbols to maintain token distinctiveness.

This method reduces vector length in word2vec while maintaining accuracy and preventing confusion. Table 2 lists the combined tokens after processing.

The results confirm that our method preserves the syntactic meaning of source code, crucial for security analysis. In contrast, `token_get_all()` only tokenizes code without considering semantic relationships.

Table 2. Combined tokens

Duplicate Tokens	**Token Combinations**
T_UNDEFINED_VAR, T_UNDEFINED_VAR, T_UNDEFINED_VAR, T_UNDEFINED_VAR	4_T_UNDEFINED_VAR
T_ECHO, T_CONSTANT_ENCAPSED_STRING; T_ECHO, T_CONSTANT_ENCAPSED_STRING;	[2_T_ECHO, T_CONSTANT_ENCAPSED_STRING;]
T_ECHO, T_CONSTANT_ENCAPSED_STRING; T_ECHO, T_CONSTANT_ENCAPSED_STRING; T_ECHO, T_CONSTANT_ENCAPSED_STRING;	[3_T_ECHO, T_CONSTANT_ENCAPSED_STRING;]
T_CONSTANT_ENCAPSED_STRING, T_CONSTANT_ENCAPSED_STRING	2_T_CONSTANT_ENCAPSED_STRING
[T_CONSTANT_ENCAPSED_STRING . ” T_CONSTANT_ENCAPSED_STRING]	[2_T_CONSTANT_ENCAPSED_STRING . ”]
T_IGNORE, T_IGNORE, T_IGNORE, T_IGNORE	4_T_IGNORE
T_SQL; T_SQL;	2_T_SQL
T_CONSTANT_ENCAPSED_STRING, T_CONSTANT_ENCAPSED_STRING, T_CONSTANT_ENCAPSED_STRING	3_T_CONSTANT_ENCAPSED_STRING
T_IGNORE, T_IGNORE, T_IGNORE	3_T_IGNORE
T_ASSIGNMENT, T_ASSIGNMENT, T_ASSIGNMENT	3_T_ASSIGNMENT
T_COMPARISON, T_COMPARISON, T_COMPARISON	3_T_COMPARISON
T_INCLUDES, T_INCLUDES	2_T_INCLUDES
T_ECHO . ” T_ECHO . ” T_ECHO . ”	3_T_ECHO . ”
T_INPUT; T_INPUT; T_INPUT;	3_T_INPUT

The `token_get_all()` function generates many tokens that do not accurately reflect function or variable usage, leading to information loss and difficulty in vulnerability detection. Our method restructures tokens to preserve meaning and relationships, enhancing deep learning model performance. Example results are shown in Figure 4.

3.7. Algorithm model

As we know, One-hot encoding is a primary embedding method that generates a sparse and discrete matrix. Essentially, this is a bag-of-words model. One-hot encoding disregards the order of words and assumes that the words are independent of each other. Word2vec is a type of word embedding method that can convert a text corpus into a vector space using neural networks. This approach can preserve semantic and syntactic relationships.

The Convolutional Neural Network (CNN) algorithm assumes that the input and output are also independent, with elements being independent of one another. CNN are commonly used in the field of image recognition. The Recurrent Neural Network (RNN), on the other hand, has an internal state to process input sequences and is well-suited for sequential data such as audio recordings and text data. However, RNNs face challenges with exploding and vanishing gradients [21].

```
1 : Source code:
2 :
3 : <?php
4 :
5 : $array = array();
6 : $array[] = 'safe' ;
7 : $array[] = $_GET['userData'] ;
8 : $array[] = 'safe' ;
9 : $tainted = $array[1] ;
10 :
11 : $tainted = (float) $tainted ;
12 :
13 : $query = "find / size '". $tainted . "'";
14 :
15 : $ret = system($query);
16 :
17 : ?>
18 :
19 :Token_get_all():
20 :
21 :T_OPEN_TAG T_VARIABLE T_ARRAY T_VARIABLE T_CONSTANT_ENCAPSED_STRING T_VARIABLE T_VARIABLE
22 :T_CONSTANT_ENCAPSED_STRING T_VARIABLE T_CONSTANT_ENCAPSED_STRING T_VARIABLE T_VARIABLE
23 : T_LNUMBER T_VARIABLE T_FLOAT_CAST T_VARIABLE T_VARIABLE T_CONSTANT_ENCAPSED_STRING
24 :T_VARIABLE T_CONSTANT_ENCAPSED_STRING T_VARIABLE T_STRING T_VARIABLE T_CLOSE_TAG
25 :
26 :PTIT:
27 :
28 :in_array ( T_INPUT [ T_CONSTANT_ENCAPSED_STRING ] , T_ARRAY ( 2_T_CONSTANT_ENCAPSED_STRING ,
29 :true ) ) http_redirect ( sprintf ( ' T_CONSTANT_ENCAPSED_STRING , T_UNDEFINED_VAR ) )
```

Fig. 4. Example of the PTIT token transformer.

The Long Short-Term Memory (LSTM) network is an improvement over RNNs for handling long-term dependencies. Within an LSTM unit, there are input, output, and forget gates designed to address the issues of exploding and vanishing gradients present in traditional RNNs. This means that LSTM models can remember more historical information and distant words compared to RNNs. Therefore, we believe that LSTM networks are more suitable for our datasets, and experience has confirmed this.

4. Experiments

4.1. Dataset

We conducted experiments based on three datasets:

- **SARD - PHP vulnerability test suite:** Bertrand Stivalet and Elizabeth Fong proposed a method for automatically generating PHP test cases. The generated PHP test cases were uploaded to SARD [22], a project of the National Institute of Standards and Technology (NIST). All 42,212 files include 29,258 safe samples and 12,954 unsafe samples in the provided datasets. The vulnerable samples contain 12 types of vulnerabilities from the Common Weakness

Enumeration (CWE), such as CWE-78: OS Command Injection, CWE-79: Cross-site Scripting, and many others. To ensure the deep learning model is provided with complete data, we had to remove three subsets with fewer than 10 samples, specifically CWE-209: Information Exposure, CWE-311: Missing Encryption of Sensitive Data, and CWE-327: Use of a Broken or Risky Cryptographic Algorithm. The detailed parameters are illustrated in Table 3.

- **SARD - PHP test suite - XSS, SQL injection 1.0.0:** Developed by Felix Schuckert and Hanno Langweg, this dataset primarily focuses on two of the most prevalent and severe vulnerabilities: CWE-79: Cross-site Scripting and CWE-89: SQL Injection. The detailed parameters are illustrated in Table 4.
- **Sqllab and XSSlab:** This application was built to train experiments on SQL Injection CWE-89 and Cross-site Scripting CWE-79 by researchers.

Table 3. Statistics of SARD - PHP vulnerability test suite

CWEs	Safe Samples	Unsafe Samples	Total
CWE-78: OS Command Injection	1872	624	2496
CWE-79: Cross-site Scripting	5728	4352	10080
CWE-89: SQL Injection	8640	912	9552
CWE-90: LDAP Injection	1728	2112	3840
CWE-91: XML Injection	4784	1264	6048
CWE-95: Eval Injection	1296	336	1632
CWE-98: PHP Remote File Inclusion	2592	672	3264
CWE-601: Open Redirect	2208	2592	4800
CWE-862: Missing Authorization	400	80	480
Total	**29248**	**12944**	**42192**

Table 4. Statistics of SARD - PHP test suite - XSS, SQL injection 1.0.0

CWEs	Quantity
CWE-89: SQL Injection	19457
CWE-79: Cross-site Scripting	24227
Safe	205185
Total	**248592**

Based on the three datasets mentioned, we aggregated samples from various sources before proceeding with the split. After aggregation, these datasets were divided into three parts: 70% for the training set, 10% for the validation set, and 20% for the test set. This approach not only optimizes the use of data from different sources, ensuring diversity and representation, but also guarantees that the model has effective real-world applicability in detecting security vulnerabilities.

4.2. Experimental environment

To test the proposed model, this paper sets up an experimental environment to evaluate accuracy, fine-tune the model, and meet application requirements. The experimental environment is configured as follows:

- Operating System: Windows 11

- Processor: Intel(R) Core(TM) CPU i5-9300H @ 2.40GHz
- Graphics Card: NVIDIA GeForce GTX 1650
- RAM: 24GB

4.3. Conducting experiments

Initially, during the experimentation process with the training model, we conducted checks, evaluations, and classifications of the collected datasets. For the SARD - PHP vulnerability test suite dataset, we divided it into 10 labels represented as a binary array in numpy, consisting of 9 labels representing specific vulnerability types and 1 safe label. For the other two datasets, the labels were categorized into two vulnerability types: XSS and SQL, along with 1 safe label. These labels were combined with the previous dataset to ensure that the model has the ability to detect new vulnerabilities, enhancing its practicality and update capability.

After transforming the source code samples with the aforementioned token technology, the word2vec model was used to convert the tokens into vector data. The model was then trained using an LSTM network, a type of recurrent neural network designed to handle sequential data while preserving long-term dependencies and avoiding the vanishing gradient problem common in traditional RNNs.

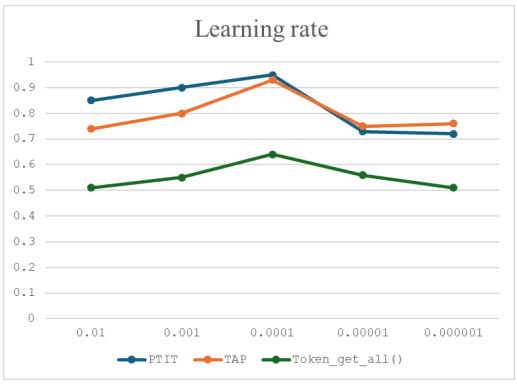
Each LSTM unit consists of three main gates: forget, input, and output. These gates act as intelligent filters, deciding which information to retain or discard at each step, enabling the network to learn effectively from sequential data.

To ensure the model operates efficiently, we meticulously adjusted the parameters. The vector size was set to 256, enabling the model to represent detailed information from the tokens. The number of LSTM units was set to 128, striking a balance between the model's learning capability and computational performance. The batch size was selected as 512, optimizing the training process, while the number of epochs was set to 60, ensuring the model had sufficient time to learn from the data without suffering from overfitting.

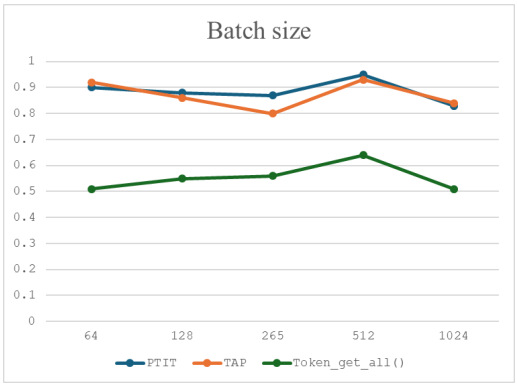
The loss function chosen was MSE (Mean Squared Error) to minimize the error between predicted values and actual values. The Adam optimizer was utilized to effectively adjust the model's weights, with a learning rate of 0.0001. This value is small enough for the model to learn steadily yet surely, helping to avoid local minima.

For the multi-class classification model, the neural network configuration was adjusted with an output layer consisting of 10 units and a softmax activation function, suitable for the classification task of 10 different labels. The loss function was also changed to categorical_crossentropy to better align with this multi-class classification problem. The diagram illustrating the experiment is shown in Figure 5.

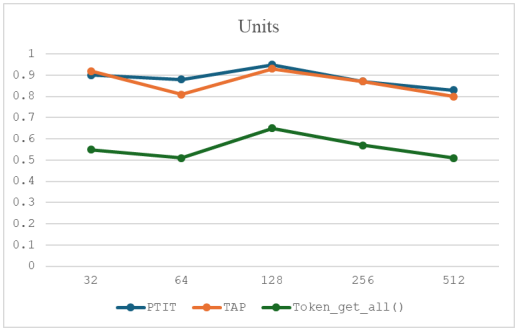
Evaluation Method: The F1 score is an important metric in evaluating security models. This metric combines both precision and recall of the classification model, helping to assess the overall performance of the model.



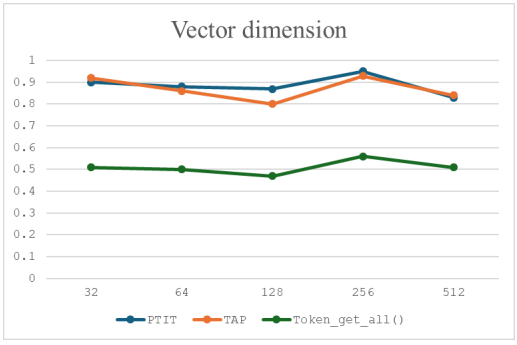
(a) Learning rate



(b) Batch size



(c) Units



(d) Vector dimension

Fig. 5. Experiments for tuning model parameters.

First, we need to understand the basic concepts:

- **True Positive (TP):** The number of samples where both the model's prediction and the actual label are positive.
- **False Positive (FP):** The number of samples where the model predicts positive, but the actual label is negative.
- **True Negative (TN):** The number of samples where both the model's prediction and the actual label are negative.
- **False Negative (FN):** The number of samples where the model predicts negative, but the actual label is positive.

The formulas for precision and recall are as follows:

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

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

Therefore, the complete formula for the F1 score is:

$$F1 = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \tag{3}$$

ROC (Receiver Operating Characteristic): A curve used to evaluate the diagnostic capability of binary classification models. The x-axis represents the False Positive Rate (FPR), and the y-axis represents the True Positive Rate (TPR). By adjusting the threshold based on the predicted probability values and the actual labels of each test sample, various points are obtained. These points can be connected to form the ROC curve. The larger the Area Under the Curve (AUC), the better the model's performance.

The formulas for TPR and FPR are:

$$TPR = \frac{TP}{TP + FN} \tag{4}$$

$$FPR = \frac{FP}{FP + TN} \tag{5}$$

4.4. Results and discussion

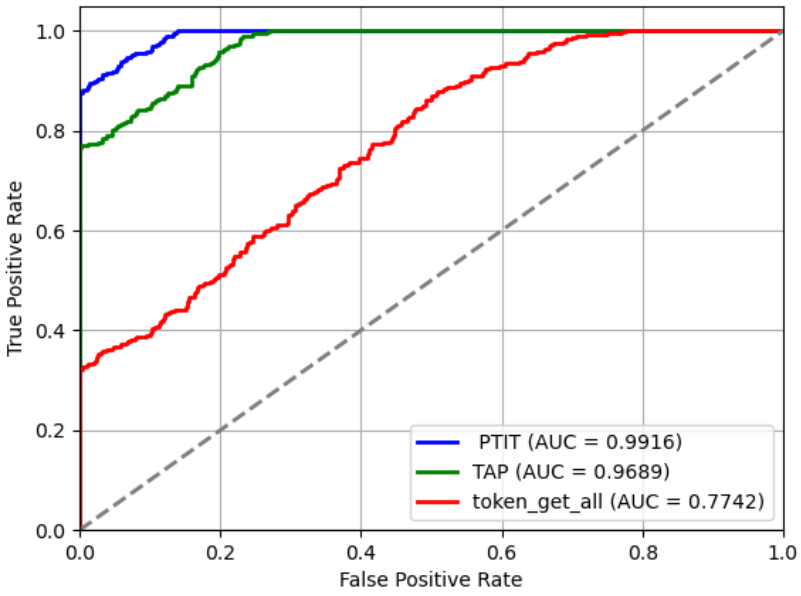


Fig. 6. Comparison of TAP, token_get_all(), and PTIT.

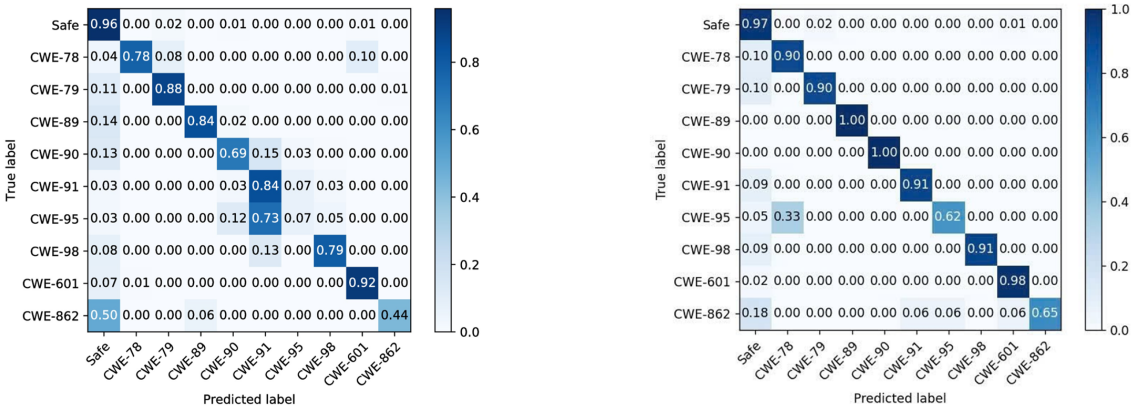
In the process of developing methods for analyzing PHP source code to detect security vulnerabilities, improving the token conversion processes plays a crucial role. One of

the most notable methods in this field is TAP [23], which was developed to overcome the limitations of PHP's standard `token_get_all()` function.

Table 5. Comparison of token analysis methods

Method	Safe Samples			Vulnerable Samples			Accuracy	AUC
	Precision	Recall	F1	Precision	Recall	F1		
PTIT	0.9812	0.961	0.97	0.91	0.85	0.88	0.9558	0.9916
TAP	0.9621	0.880	0.92	0.87	0.75	0.81	0.9356	0.9681
token_get_all()	0.9047	1.0000	0.9499	1.0000	0.0761	0.1415	0.8776	0.7654

The `token_get_all()` function initially simply generates a basic set of tokens from PHP source code, but it cannot distinguish specific functions or provide in-depth information about the code structure. TAP has significantly improved this process by constructing specialized tokens for each function in the PHP source code, thereby enhancing the analysis and vulnerability detection capabilities. The TAP method achieved an accuracy of 0.9356, a significant advancement compared to solely using the initial `token_get_all()` function.



(a) Normalized confusion matrix of TAP. *(b) Normalized confusion matrix of PTIT.*

Fig. 7. Comparison of normalized confusion matrices between TAP and PTIT.

However, the proposed model outperforms TAP and `token_get_all()`, achieving an accuracy of 0.9558, higher than TAP. This improvement enhances both safe sample identification and complex vulnerability detection. The model effectively handles highly duplicated source code while preserving semantics, overcoming previous limitations. Training results are detailed in Table 5, with the ROC curve shown in Figure 6, affirming the model's practical applicability in vulnerability detection.

To highlight its multi-class classification capability, we compared the model using the SARD dataset against TAP (Figure 7).

Results show significant improvements over previous models, particularly in CWE-89 (SQL Injection) and CWE-90 (Improper Neutralization of Special Elements), exceeding expectations.

Additionally, our model reduces confusion in classifying CWE-91 (XML Injection) and CWE-95 (Improper Neutralization of Special Elements), where older models struggled. These improvements enhance detection accuracy, demonstrating the model's superiority in handling complex vulnerabilities.

5. Conclusion

This paper presents an advanced model for source code analysis, combining source code transformation into tokens to enhance security vulnerability detection. Specifically, the proposed model utilizes parsing techniques to convert source code into basic units called tokens, which are then analyzed using advanced machine learning methods for effective detection of security vulnerabilities.

The development process involves converting source code into accurate tokens, retaining important structural and semantic information. These tokens are then input into machine learning models to identify security weaknesses. Experimental results show this model not only achieves superior performance compared to existing methods but also significantly improves accuracy. Compared to previously published models, it demonstrates clear superiority in vulnerability detection, thanks to the integration of token transformation and machine learning techniques.

However, the proposed method has limitations. First, the model relies on existing source code samples, so its effectiveness may decrease when applied to new or untested code. Second, converting source code into tokens can be challenging with complex or non-standard syntax. Additionally, training deep models like LSTM requires significant computational resources, complicating deployment on large or real-time systems.

Future work will focus on enhancing generalization to diverse source code and optimizing token conversion for complex snippets. Investigating convolutional layers will improve pattern detection, while developing lighter models will reduce computational costs and training time, facilitating deployment in larger or real-time systems.

6. Acknowledgement

This research is funded by the project "Research on developing technical regulations on testing and evaluating information security for web-based applications of Sector of Natural Resources and Environments," code TNMT.2023.04.01.

References

[1] W.R. Cheswick, S.M. Bellovin, and A.D. Rubin, "Firewalls and Internet Security: Repelling the Wily Hacker," 2rd endition, Addison-Wesley Longman Publishing Co., Inc., 2003.

[2] Y. Wang, Y. Shen, H. Wang, J. Cao, and X. Jiang, "MtMR: Ensuring MapReduce computation integrity with Merkle Tree-Based verifications," IEEE Transactions on Big Data, vol. 4, no. 3, 2016, pp. 418–431. DOI: 10.1109/TBDATA.2016.2599928

[3] J. Shu, X. Jia, K. Yang, and H. Wang, "Privacy-preserving task recommendation services for crowdsourcing," IEEE Transactions on Services Computing, vol. 11, no. 5, 2018, pp. 828–841. DOI: 10.1109/TSC.2018.2791601
[4] N.T. Cong, L.H. Toan, and T.M. Thanh, "An Overview of Static and Dynamic Analysis in Application Security Testing," *Journal of Military Science and Technology*, vol. 99, no. 99, 2024, pp. 1-11. DOI: 10.54939/1859-1043.j.mst.99.2024.1-11
[5] J. Dahse and J. Schwenk, "RIPS—A static source code analyser for vulnerabilities in PHP Scripts," Seminar Work (Seminer C¸ alismasi), Horst Go¨rtz Institute Ruhr-University Bochum, 2010.
[6] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006, pp. 6–pp. DOI: 10.1109/SP.2006.29
[7] S. Son and V. Shmatikov, "SAFERPHP: Finding semantic vulnerabilities in PHP applications," in *Proceedings of the 2011 ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security, ACM*, 2011, pp. 8. DOI: 10.1145/2166956.2166964
[8] F. Tip, "A survey of program slicing techniques," *Computer Science and telematics*, 1994.
[9] J. Dahse, N. Krein, and T. Holz, "Code reuse attacks in PHP: Automated pop chain generation," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, ACM*, 2014, pp. 42–53. DOI:10.1145/2660267.2660363
[10] R. Russell, L. Kim, L. Hamilton *et al.*, "Automated Vulnerability Detection in Source Code Using Deep Representation Learning," in *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2018, pp. 757–762. DOI: 10.1109/ICMLA.2018.00120
[11] J.C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, 1976, pp. 385–394. DOI: 10.1145/360248.360252
[12] S. Lawrence, C.L. Giles, A.C. Tsoi, and A.D. Back, "Face recognition: A convolutional neural-network approach," *IEEE Transactions on Neural Networks*, vol. 8, no. 1, pp. 98–113, 1997. DOI: 10.1109/72.554195
[13] Z. Li, D. Zou, S. Xu *et al.*, "VulDeePecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018. DOI:10.48550/arXiv.1801.01681
[14] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional LSTM and other neural network architectures," *Neural Networks*, vol. 18, no. 5-6, pp. 602–610, 2005. DOI: 10.1162/neco.1997.9.8.1735
[15] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan, "NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security 18)*, pp. 377–392, 2018.
[16] A. Doupe, B. Boe, C. Kruegel, and G. Vigna, "Fear the EAR: discovering and mitigating execution after redirect vulnerabilities," in *Proceedings of the 18th ACM conference on Computer and Communications Security*, ACM, 2011, pp. 251–262. DOI: 10.1145/2046707.2046736
[17] PHP, "token_get_all—Manual," *PHP.net*, 07 Sept, 2006. [Online]. Available: http://www.php.net/token-get-all. [Accessed: 25 April 2024].
[18] PHP, "List of Parser Tokens—Manual," *PHP.net*, 07 Sept, 2010. [Online]. Available: http://php.net/manual/en/tokens.php. [Accessed: 25 April 2024].
[19] T. Mikolov, I. Sutskever, K. Chen, G.S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems*, 2013, pp. 3111–3119. DOI: 10.48550/arXiv.1310.4546
[20] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997. DOI: 10.1162/neco.1997.9.8.1735
[21] W. Zaremba, I. Sutskever, and O. Vinyals, "Recurrent neural network regularization," *arXiv preprint arXiv:1409.2329*, 2014. DOI: 10.48550/arXiv.1409.2329
[22] B. Stivalet and E. Fong, "Large scale generation of complex and faulty PHP test cases," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2016, pp. 409–415. DOI: 10.1109/ICST.2016.43
[23] Y. Fang, S. Han, C. Huang, and R. Wu, "TAP: A static analysis model for PHP vulnerabilities based on token and deep learning technology," *PLOS ONE*, vol. 14, no. 6, 2019. DOI: 10.1371/journal.pone.0226887

Manuscript received 22-09-2024; Accepted 27-12-2024.

■

Van Cong Nguyen is a cadet of Le Quy Don Technical University in Vietnam. His research interests lie in the area of deep learning, security holes detection, and computer vision.
E-mail: congnguyen.1801b@gmail.com

Huy Toan Le is a leader of Center for Information Technology Product Testing – Department of Digital Transformation and Environment Resources Data Information – Ministry of Natural Resources and Environment, 28 Pham Van Dong, Cau Giay, Hanoi, Vietnam. His research interests lie in the area of deep learning, security holes detection, and computer vision.
E-mail: lhtoan@monre.gov.vn

Minh Thanh Ta is currently an associate professor and vice dean of Institute of Information and Communication Technology in Le Quy Don Technical University, Vietnam. He is also a Postdoctoral Fellow of the Department of Mathematical and Computing Sciences at Tokyo Institute of Technology. He received his B.S. and M.S in Computer Science from National Defense Academy, Japan, in 2005 and 2008 and his Ph.D. from Tokyo Institute of Technology, Japan, in 2015, respectively. He is a member of IPSJ Japan and IEEE. His research interests lie in the area of watermarking, network security, and computer vision.
E-mail: thanhtm@lqdtu.edu.vn

PHÂN LOẠI LỖ HỔNG CVE TRONG MÃ NGUỒN DỰA TRÊN PHÂN TÍCH TOKEN KẾT HỢP MẠNG LSTM

Nguyễn Văn Công, Lê Huy Toàn, Tạ Minh Thanh

Tóm tắt

Khi các ứng dụng web ngày càng trở nên phổ biến, tầm quan trọng của bảo mật mã nguồn đang tăng lên nhanh chóng. Các lỗ hổng bị lộ gây ra rủi ro nghiêm trọng cho cả nhà cung cấp dịch vụ và khách hàng. Nhiều mô hình khác nhau đã được đề xuất để giải quyết vấn đề này; tuy nhiên, hầu hết các phương pháp đều dựa vào các cấu trúc đồ thị phức tạp được tạo từ mã nguồn hoặc trên các mẫu biểu thức chính quy do chuyên gia điều khiển. Bài báo này giới thiệu một mô hình sử dụng các cơ chế dựa trên mã thông báo kết hợp với các kỹ thuật học sâu để phát hiện lỗ hổng hiệu quả trong các ứng dụng web PHP (Bộ xử lý siêu văn bản). Bằng cách tận dụng quy trình mã thông báo PHP, chúng tôi đã phát triển một mã thông báo tùy chỉnh hợp nhất các mã thông báo, hỗ trợ các tính năng PHP chính và tối ưu hóa việc phân tích cú pháp. Sử dụng các tập dữ liệu như Bộ dữ liệu tham chiếu đảm bảo phần mềm (SARD) và SQLI-LABS, bài báo này trình bày quá trình đào tạo mô hình học sâu với các mã thông báo nâng cao để phát hiện hiệu quả các lỗ hổng trong mã nguồn.

Từ khóa

Mã thông báo; mã nguồn; học sâu; phát hiện lỗ hổng; xử lý ngôn ngữ tự nhiên; mã nguồn PHP.