VulHunter: An Automated Vulnerability Detection System Based on Deep Learning and Bytecode

Ning Guo[1], Xiaoyong Li[1(✉)], Hui Yin[2], and Yali Gao[1]

[1] Key Laboratory of Trustworthy Distributed Computing and Service (BUPT), Ministry of Education, Beijing, China
lixiaoyong@bupt.edu.cn

[2] College of Computer Science and Engineering, Shandong University of Science and Technology, Qingdao, China

Abstract. The automatic detection of software vulnerability is undoubtedly an important research problem. However, existing solutions heavily rely on human experts to extract features and many security vulnerabilities may be missed (i.e., high false negative rate). In this paper, we propose a deep learning and bytecode based vulnerability detection system called Vulnerability Hunter (VulHunter) to relieve human experts from the tedious and subjective task of manually defining features. To the best of knowledge, we are the first to leverage bytecode features to represent vulnerabilities. VulHunter uses the bytecode, which is the intermediate representation output by the source code, as input to the neural networks and then calculate the similarity between the target program and vulnerability templates to determine whether it is vulnerable. We detect SQL injection and Cross Site Scripting (XSS) vulnerabilities in PHP software to evaluate the effectiveness of VulHunter. Experimental results show that VulHunter achieves more than 88% (SQL injection) and 95% (XSS) F1-measure when detecting a single type of vulnerability, as well as more than 90% F1-measure when detecting mixed types of vulnerabilities. In addition, VulHunter has lower false positive rate (FPR) and false negative rate (FNR) than existing approaches or tools. In practice, we apply VulHunter to three real PHP software (SEACMS, ZZCMS and CMS Made Simple) and detect five vulnerabilities in which three have not been disclosed before.

Keywords: Vulnerability detection · Deep learning · Bytecode

1 Introduction

Nowadays, nearly all information systems and business applications have built software-based applications. However, because of their existing security vulnerabilities which may be uncovered and unexploited, software are exposed to attacks, which will have a highly negative impact on users. According to the

© Springer Nature Switzerland AG 2020
J. Zhou et al. (Eds.): ICICS 2019, LNCS 11999, pp. 199–218, 2020.
https://doi.org/10.1007/978-3-030-41579-2_12

2018 Application Security Statistics Report from WhiteHat [3], more than 60% of applications have long been in a vulnerable environment. Thus, security experts have developed various solutions to detect vulnerabilities quickly and efficiently.

Vulnerability detection technology can be divided into three types depending on whether the program is running or not: dynamic analysis, static analysis, and mixed analysis [27]. The dynamic analysis method examines a program's behavior while it is running in a given environment. Essentially, dynamic analysis adopts an approach similar to that of a real attacker, and many commercial and open-source tools [2,4,15] and studies [16–19] have been proposed. The static analysis method detects program vulnerabilities without executing it. Usually, the static method analyzes the control logic and data flow of the program, and combines data statistics and feature recognition to determine whether the program has a vulnerability. Many systems and studies for this purpose have been conducted, including open source tools [1,10], commercial tools [5,9], and academic research projects [23,28,32,33]. Mixed analysis is a combination of dynamic and static analysis.

However, existing static solutions for vulnerability detection demonstrate over-reliance on expert experience and extract only surface source code features. The features of the source code are strongly related to the writing style of the code. The same vulnerability may appear in different source code, resulting in poor generalization of the neural networks. To address these technical challenges, we propose a vulnerability detection system called Vulnerability Hunter (VulHunter), which uses deep learning to calculate the similarity of bytecode features. We use bidirectional LSTM to build a neural network that can input two vectors and output a similarity value. We use graph-based static analysis methods to extract the code slices associated with the vulnerabilities and then transform them into bytecode slices, which are a lower-level representation of the code. In Sect. 4.3, we further demonstrate the effectiveness of the bytecode through experiments and explain it. We use PHP as a sample language to demonstrate the effectiveness of our proposed approach. The PHP source code is split into bytecode slices and word2vec [14] is used to transform them into vectors that can be fed to neural networks. The vulnerability templates are bytecode slices that have been processed, and each template represents a specific vulnerability. There are multiple templates for each vulnerability. The trained neural networks calculate the similarity between the target program and vulnerability templates to determine whether there are any vulnerabilities. In the experiment, we mainly detect SQL injection and XSS vulnerabilities, and there are 160 and 208 templates for the two vulnerabilities respectively. To verify the effectiveness of the system, we evaluate VulHunter on three open source PHP software: SEACMS, ZZCMS and CMS Made Simple. Five vulnerabilities were successfully detected, and three of them were discovered for the first time.

In summary, the contributions of this paper can be highlighted as follows.

– To the best of our knowledge, we are the first to apply bytecode to represent vulnerability features and employ graph-based static analysis methods to extract bytecode slice, which is the smallest unit of vulnerability

representation. The bytecode slice is converted from a few lines of source code associated with the vulnerability to accurately represent and locate the vulnerability.
- We propose a deep learning based approach to detect vulnerabilities. Different from other existing studies which directly determine whether the target program has a vulnerability, our neural networks are mainly leveraged to calculate the similarity of the target program and the vulnerability templates. When the value of similarity exceeds a certain threshold, it will determine that the target program is vulnerable, which effectively improves the accuracy and recall rate of the system.
- A comprehensive experimental study on three real sample collections is performed to compare with the state-of-art vulnerability detection approaches. The promising experimental results demonstrate that our developed system VulHunter which integrate our proposed method outperforms other alternative vulnerability detection techniques. The code, data sets and vulnerability templates of this work is publicly available at https://github.com/Xmansec/VulHunter.

The remainder of this paper is organized as follows. In Sect. 2, we provide background on software vulnerability detection. In Sect. 3, we detail our methodology for detecting vulnerabilities and our improved technique for representing vulnerable features. In Sect. 4 we describe the results of our experiments for detecting SQL injection and XSS vulnerabilities in PHP software. At the same time, we show results of comparison with other approaches and tools. In Sect. 5, we introduce some previous work related to this paper. Finally, we conclude the paper in Sect. 6.

2 Background

Here, we provide the background relevant to software vulnerability detection. First, we give some examples of SQL injection and XSS vulnerabilities in Sect. 2.1. Then, In Sect. 2.2, we introduce the graph-based static software analysis methods. In Sect. 2.3, we discuss bytecode and its component in PHP. Finally, in Sect. 2.4, we cover background on Bi-LSTM neural networks.

2.1 SQL Injection and XSS Vulnerabilities

In this paper, we use SQL injection and XSS vulnerabilities to verify the effectiveness of the system. SQL injection refers to a class of code-injection attacks in which data provided by the user is included in an SQL query in such that part of the user's input is treated as SQL code [24]. Figure 1(a) shows an SQL injection vulnerability example in PHP source code. In the example, an attacker can craft a link that paired the single quotes of *id* and injects an arbitrary SQL such as http://www.xxx.com/?id=-1' union select database() –+. With this link, the attacker may obtain the name of the database. Cross-Site Scripting (XSS)

```
1 <?php
2 $id=$_GET['id'];
3 $sql="SELECT * FROM users WHERE id='$id'";
4 // exploit
5 $result=mysql_query($sql);
6 $row = mysql_fetch_array($result);
7 echo $row;
8 ?>
```

(a) SQL injection

```
1 <?php
2 $data = $_GET['id'];
3 $data = urldecode($data);
4 //exploit
5 echo "<div ". $data ."= bob />" ;
6 ?>
```

(b) XSS

Fig. 1. Examples of SQL injection and XSS vulnerability.

is one of the ten most critical web software security risks. There are three forms of XSS: reflected, stored and DOM XSS. Figure 1(b) shows a reflected XSS vulnerability example in PHP source code. In the example, an attacker can craft a link that injects malicious HTML and JavaScript code into the front page such as *http://www.xxx.com/?id=*><script>*alert(document.cookie)*</script><. This link pops up a window showing the *cookies* of the current website.

2.2 Graph-Based Static Analysis

Graph-based static analysis refers to modeling program properties as graphs such as control-flow graphs (CFG), data-flow graphs (DFG) and program-dependence graphs (PDG) [27]. These techniques rely on building a model of bugs by a set of nodes in the graphs to identify bugs in a program. In this paper, we use a graph-based static analysis methods for code slicing, which is an important step in data processing. There are already some well-known algorithms to implement it [26,30]. These methods can help us analyze the program, build a static graph of the program, and extract the code related to the vulnerability. Therefore, we do not need to analyze the entire program file, and can more accurately represent and locate the vulnerability.

2.3 Bytecode

Bytecode is a form of instruction set designed for efficient execution by a software interpreter. It is an intermediate representation output by programming language implementations to ease interpretation or to reduce hardware and operating system dependence by allowing the same code to run cross-platform. Bytecode often directly executes on a virtual machine that further compiled bytecode into machine code for improving performance. Thus, the bytecode is a kind of code between the source code and the machine code, which can represent the semantic information of the code at a lower-level. Given its performance advantage, many languages first convert the source code into bytecode and then transform it into machine code or execute it directly in the virtual machine, such as Java, Python, and PHP.

2.4 Bi-LSTM

Many different types of neural networks have been successfully applied in numerous fields. The neural networks used in VulHunter are Bidirectional Long Short-Term Memory (Bi-LSTM) networks, which is a type of Recurrent Neural Networks (RNN). The problem of vanishing gradients is a key motivation behind the application of the LSTM cell [20,21,25], which consists of a state that can be read, written, or reset via a set of programmable gates. The multiplicative gates allow LSTM memory cells to store and access information over long time periods, thereby mitigating the vanishing gradients problem. For example, as long as the input gate remains closed (i.e., it has an activation near 0), the activation of the cell will not be overwritten by the new inputs arriving in the network and can be made available to the net much later in the sequence by opening the output gate [22]. However, even LSTM is insufficient for vulnerability detection because the argument(s) of a program may be affected by earlier or later statements. This result suggests that unidirectional LSTM can not learn enough vulnerability features and that we should use Bi-LSTM. Figure 5 shows a brief structure of neural networks in VulHunter.

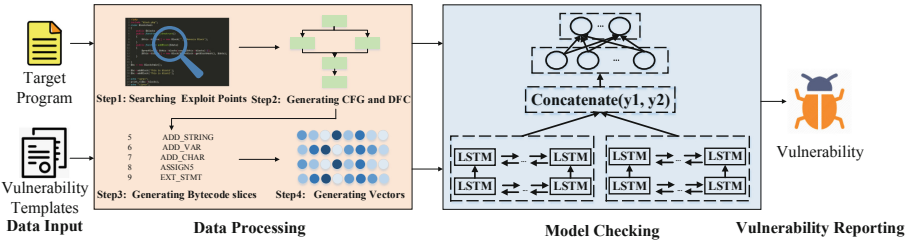

Fig. 2. Overview of VulHunter. It has two inputs, four data processing steps, one model checking step, and one output.

3 Methodology

In this section, we describe the methodology for our system to detect vulnerabilities (SQL injection and XSS) in PHP source code. In Sect. 3.1, we give an overview of VulHunter for vulnerability detection and elaborate its steps. In Sect. 3.2, we describe how to locate suspicious exploit points and generate bytecode slices from source code. In Sect. 3.3, we discuss the method to transform bytecode slices into vectors. In Sect. 3.4 we describe how to use Bi-LSTM networks in VulHunter. In Sect. 3.5, we present a formula for similarity calculation.

3.1 Overview of VulHunter

This subsection is an overview of VulHunter. Figure 2 shows the steps of methodology. VulHunter converts target program, along with the vulnerability templates, to digital vector during data processing phase. The neural networks calculate the similarity between the vulnerability templates and target program in

model check phase. When the value of the similarity exceeds a certain threshold, the target program is deemed vulnerable. The following is a detailed process of the vulnerability detection.

Data Processing

- Step 1: Search for exploit points from source code, which are usually some functions or specific code (HTML tags). These points are necessary but not sufficient conditions for the existence of vulnerabilities.
- Step 2: Based on the exploit points found in Step 1, we generate CFG and DFG. Usually the input is the beginning of the graph, and the output is the end of the graph. Some of the edges in the figure are the path of the vulnerability execution.
- Step 3: According to the graph generated by Step 2, we use graph-based static analysis methods to generate code slices, which are just some code related to suspicious exploit points, such as variable declarations, function calls. And then we transform code slices to bytecode slices. Bytecode slices are the smallest unit that represents a vulnerability in this paper.
- Step 4: Bytecode slices generated by Step 3 are split into tokens by a tokenizer and then transform them into digital vectors of length L and pre-train with word2vec. This tool is based on the idea of distributed representation, which maps a token to an integer.

Model Checking. Input the target vector obtained in the previous step and vulnerability templates into the trained neural networks for similarity calculation. Finally, use the value of the similarity to determine whether it is vulnerable.

3.2 Generating Bytecode Slices

In this subsection, we describe the details how to convert the PHP source code into bytecode slices that corresponds to step 1 to step 3 of Fig. 2. A program file

Table 1. Suspicious exploit points related to SQL injection and XSS. The suspicious exploits of the SQL injection vulnerability are mainly functions that execute SQL, and the XSS vulnerability is related to various *HTML tags* (e.g., *<a>*, * *) in addition to output functions.

Vulnerability	Suspicious exploit points
SQL injection	mysql_connect mysql_pconnect mysql_change_user mysql_query mysql_error mysql_set_charset mysql_unbuffered_query pg_connect pg_pconnect pg_execute pg_insert pg_put_line pg_query pg_select pg_send_query pg_set_client_encoding pg_update sqlite_open sqlite_poen sqlite_query sqlite_array_query sqlite_create_function sqlite_create_aggregate sqlite_exec mssql_connect mssql_query sqlsrv_connect sqlsrv_query odbc_connect odbc_exec
XSS	echo print printf print_r var_dump *HTML tags*

has numerous lines of code, but only a small part is related to the vulnerability. Thus, we need to use a more concise and accurate method to represent the vulnerability, not the entire file.

Searching Exploit Points. To generate bytecode slices, we must first examine for exploit points. The point referring to the vulnerability are finally triggered. It may be some functions or specific code (HTML tags). In the examples of this paper, we focus primarily on functions related to SQL execution and front-end display, such as *mysql_query()* and *echo* in Fig. 1. Table 1 summaries the suspicious exploit points related to SQL injection and XSS vulnerabilities in PHP.

Generating CFG and DFG. Code slices consists of a number of lines of code generating by graph-based static analysis methods. We construct graph based on control flow and data flow. The input is the beginning of the graph, and the output is the end. There are usually several methods to input data. First, data can be obtained from HTTP(S) communication; second, data are from the files; finally, data are from the database. For the first case, HTTP(S) methods are generally used, such as GET, POST and PUT. For the second case, it is generally related to the file uploaded by the user. In the third case, the data of the user has usually been stored in the database, which is more likely to cause secondary SQL injection or stored XSS. Figure 3 shows the details of the whole process. The source code is represented as a structured graph, then the execution path related to the vulnerability is determined. Finally, the code on the execution path is extracted.

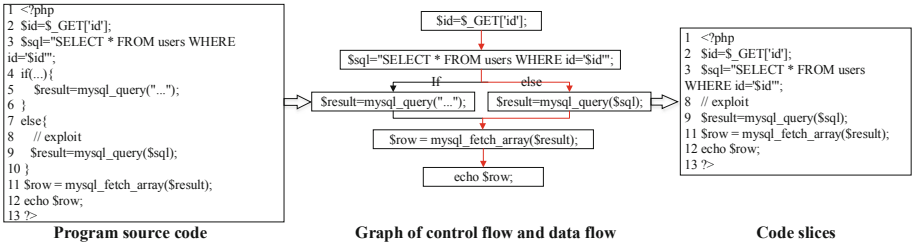

Fig. 3. Modified example of Fig. 1(a), in which the program has a SQL injection vulnerability. In this example, the code slices consists of five statements, namely lines 2, 3, 9, 11 and 12 of the program. When the program executes the *$result=mysql_query($sql)* in *else* code block, the vulnerability would be exploited, which is indicated by the red arrow. We extract these lines of the program to generate code slices. (Color figure online)

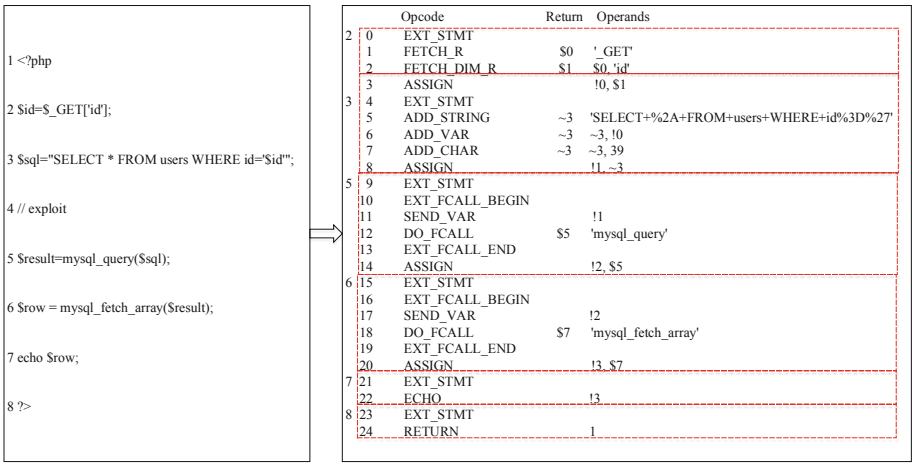

Fig. 4. Code slices and its corresponding bytecode slices. Each line of source code corresponds to several lines of bytecode. The variable name is automatically replaced, for example: !0 = $id, !1 = $sql, !2 = $result, !3 = $row.

Generating Bytecode Slices. In this step, we transform the code slices into bytecode slices by using VLD [13]. Although there are certain specifications and standards for programming, different programmers still have varying programming habits. In other words, the same vulnerability is likely to have multiple different representation at the source code level. Learning common vulnerability features is difficult from the perspective of source code. Bytecode is an more abstract representation of program. Neural networks can learn lower-level features to avoid overfitting. Figure 4 shows code slices (showed in Fig. 3) and its corresponding bytecode slices.

3.3 Transforming Bytecode Slices into Vectors

In this section, we transform the bytecode slices into digital vectors that corresponds to step 4 of Fig. 2. Given that bytecode conversion has completed tasks, such as removing comments and replacing variable names, excess data preprocessing is unnecessary. However, some operands are URL-encoded (e.g., third line of Fig. 4), so they are decoded first. Each bytecode slice needs to be encoded into a digital vector. For this purpose, we transform bytecode slices into token sequence, and each opcode follows the corresponding operations. All tokens would be split by space. For example, third line of Fig. 4:

$$\$sql = \text{“}SELECT\ *\ FROM\ users\ WHERE\ id = \text{‘}\$id\text{’ ”};$$

Whose bytecode slices are

```
4   EXT_STMT
5 ADD_STRING ~ 3 'SELECT+%2A+FROM+
                  users+WHERE+id%3D%27'
6   ADD_VAR   ~ 3 ~ 3, !0
7  ADD_CHAR   ~ 3 ~ 3, 39
8    ASSIGN       !1, ~ 3
```

They can be represented by a sequence of 17 tokens:

"EXT_STMT", "ADD_STRING", "SELECT", "", "FROM", "users", "WHERE", "id", "ADD_VAR", "~ 3", "!0", "ADD_CHAR", "~ 3", "39", "ASSIGN", "!1", "~ 3"*

This practice leads to a large corpus of tokens. In order to transform these tokens into digital vectors, we use tokenizer to encode each token into a unique number, which represents the position of token in the entire vector space.

Bytecode slices may have different numbers of tokens, so that corresponding digital vectors may have different length. Bi-LSTM takes equal-length vectors as input, so we need to make an adjustment. For this purpose, we introduce a parameter L as the fixed length of vectors corresponding to bytecode slices. There are two cases: when a vector is shorter than L, we pad zeros in the beginning of the vector; when a vector is longer than L, we delete the beginning part of the vector. We first determine the suspicious exploit points and then track the input forward, so the later tokens are crucial. This parts of the vector are retained when padding and deleting.

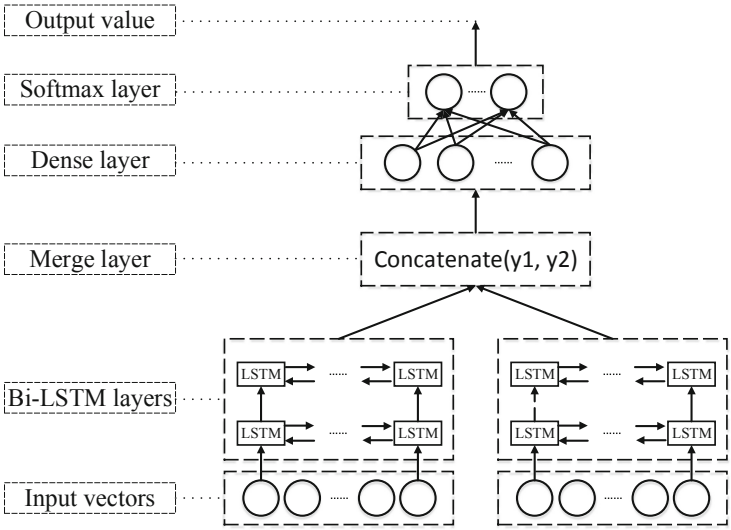

Fig. 5. A brief structure of Bi-LSTM neural networks. It evaluates the similarity of two vectors and products the result. The Bi-LSTM layers can propagate errors both forward and backward to avoid vanishing gradients problem.

3.4 Building Bi-LSTM Neural Networks

The core of our method is to calculate the similarity of bytecode between vulnerability templates and target program. Therefore, neural networks need to be able to receive two inputs and produce one output. Figure 5 shows the structure of neural networks, which has a number of Bi-LSTM layers, a merge layer, a dense layer, and a softmax layer. The Bi-LSTM layers contain some complex LSTM cells, with two directions (forward and backward). The merge layer concatenates the outputs of the two Bi-LSTM layers and combines them into a single tensor. The dense layer reduces the number of dimensions of the vectors received from the Bi-LSTM layers. The softmax layer takes the low-dimension vectors received from the dense layer as input, and is responsible for representing and formatting the classification result, which provides feedback for updating the neural network parameters in the learning phase.

3.5 Similarity Calculation

The same types of vulnerabilities may have many different subtypes, which indicate various causes of the vulnerabilities. Therefore, we use a vulnerability template to represent a subtype of vulnerability. The target program and vulnerability templates are then used as inputs of the neural networks to calculate the mean of similarity, denoted as S:

$$S = \frac{\sum_{i=1}^{N} BLNN(T_i, P)}{N} \tag{1}$$

where N is the number of vulnerability templates, $BLNN$ is Bi-LSTM neural networks, T_i is a template for a subtype of a vulnerability, and P is target program. The output of $BLNN$ is number between 0 (dissimilar) and 1 (similar). When the value of S exceeds a certain threshold (we set it to 0.5), the target program is considered to be vulnerable.

4 Experiments and Results

In the following section, we detail of our experimental steps and results. In Sect. 4.1, we give the evaluation metrics. In Sect. 4.2, we describe the process of data preprocessing and model training. In Sect. 4.3, we use VulHunter to detect signal or mixed vulnerabilities and compare results with other methods and tools. Final, we use VulHunter in practice.

4.1 Evaluation Metrics

Precision, recall, F1-measure, false negative rate (FNR), and false positive rate (FPR) are the five metrics to evaluate vulnerability detection system. A confusion matrix is used to calculate these parameters. In the confusion matrix, true positive (TP) is the number of samples with vulnerabilities detected correctly.

False positive (FP) is the number of samples without vulnerabilities detected incorrectly. True negative (TN) is the number of samples without vulnerabilities undetected. False negative (FN) is the number of samples with vulnerabilities undetected.

- Precision (P) shows how many vulnerabilities detected by system are actual vulnerabilities. The higher P is, the lower false alarm is:

$$P = \frac{TP}{TP + FP} \tag{2}$$

- Recall (R) shows the percentage of detected vulnerabilities versus all vulnerabilities presented. We want a high R value:

$$R = \frac{TP}{TP + FN} \tag{3}$$

- F1-measure (F1) is the harmonic mean of precision and recall. We also aim for a high F1-measure value:

$$F1 = 2\frac{P \cdot R}{P + R} \tag{4}$$

- The FNR measures the ratio of false negative vulnerabilities to the entire population of samples that are vulnerable. We want a low FNR value:

$$FNR = \frac{FN}{FN + TP} \tag{5}$$

- The FPR measures the ratio of false positive vulnerabilities to the entire population of samples that are not vulnerable. We also want a low FPR value:

$$FPR = \frac{FP}{FP + TN} \tag{6}$$

4.2 Data Preprocessing and Model Training

Description of Datasets. Our experiments are mainly based on SQL injection and XSS vulnerabilities numbered CWE-89 and CWE-79 in common weakness enumeration (CWE) [6], which is a community-developed list of common software security weaknesses. The PHP code data comes mainly from NVD [7], which contains vulnerability programs in production software, and SARD [11], which has many vulnerability cases. In NVD and SARD, each program or case has a CWE ID that indicates which type of vulnerability it belongs to. In total, we collected 18,989 programs, of which 912 have SQL injection vulnerabilities and 4,352 have XSS vulnerabilities. The rest are the invulnerable programs without a known vulnerability, of which 7,992 are related to SQL injection and 5,733 are related to XSS. We used ten-fold cross-validation, using 90% of the data as a training set and 10% as a test set. The PHP source code obtained from NVD is vulnerable, and the programs obtained from SARD have been marked as “bad” or “good”. Thus we do not need to manually mark them as vulnerable or invulnerable.

Table 2. Statistics of the three datasets

Dataset	Bytecode slices	Vulnerable bytecode slices	Invulnerable bytecode slices
SQL-SET	8904	912	7992
XSS-SET	10085	4352	5733
MIX-SET	18989	5264	13725

Data Preprocessing. We transform source code into bytecode slices according to steps 1 to 3 in 3.1 selection and then divide the dataset into the following three parts: SQL injection vulnerability set, XSS vulnerability set, and mixed types of vulnerabilities set (i.e., the sum of the first two sets). Table 2 summarizes the number of bytecode slices in datasets. In the SQL-SET dataset, there are 1,032,509 tokens, of which 182 are different; in the XSS-SET dataset, there are 510,252 tokens, of which 168 are different. The MIX-SET is sum of SQL-SET and XSS-SET. These symbolic representations are encoded into digital vectors for training neural networks.

Training Neural Networks. We use the three datasets in Table 2 to train neural networks and find the best parameters. The training process of the four main parameters of *number of Bi-LSTM layer*, *batch_size*, *dropout* and *epoch* is shown in Fig. 6. The selection of each parameter mainly refers to the F1-measure value and the training time. The more the number of Bi-LSTM layers, the more complex the neural networks are, which may increase the system's performance and uptime. Figure 6(a) shows the process of changing the F1-measure and number of seconds pre epoch as the number of Bi-LSTM layers increases. When the number of Bi-LSTM layers is 2, the F1-measure is the highest and the number of seconds per epoch is also low, so the number of Bi-LSTM layers is set to 2. Similarly, batch_size is set to 256, dropout is set to 0.2, and epoch is set to 20.

4.3 Vulnerability Detection

In this selection, we verify the vulnerability detection capability of VulHunter and compare with other vulnerability detection approaches or tools. We select three open source tools called RIPS [9], sonarqube [12], phpcs-security-audit [8], and a state-of-the-art systems called VulDeePecker [28]. RIPS, sonarqube, and phpcs-security-audit are all PHP vulnerability detection tools recommended by OWASP and widely used. VulDeePecker is a deep learning-based vulnerability detection system proposed by Li et al. [28]. RIPS, sonarqube, and phpcs-security-audit can directly detect PHP source code. VulDeePecker was originally designed to detect buffer error vulnerability and resource management error vulnerabilities in C/C++ programs. For a fair comparison, we construct neural networks

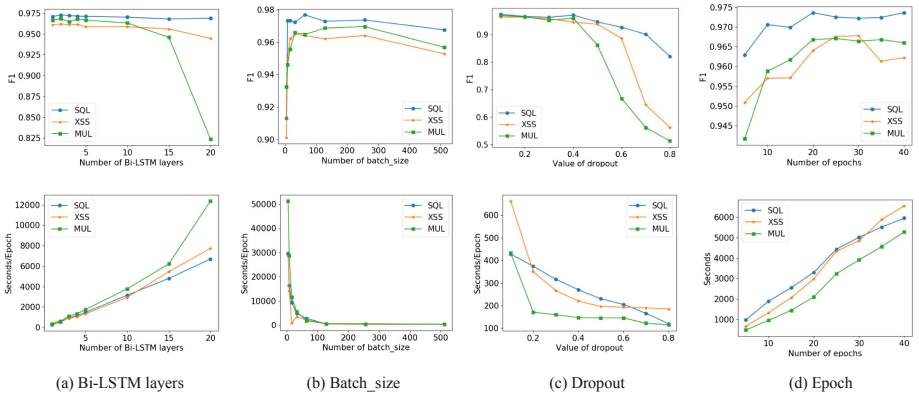

Fig. 6. Parameter selection. This picture consists of four parts, (a) number of Bi-LSTM layer, (b) batch_size, (c) dropout, and (d) epoch, which all show the trend of F1- measure and time as parameters change. The time of (a), (b), and (c) is the number of seconds per epoch, and the time of the last part (d) is the number of seconds that all epochs add up.

according to VulDeePecker's design steps and retrain by our datasets. Table 3 shows the detection results of the system.

First, Let's analyze the results of Vulhunter. Its performance is better at detecting XSS vulnerabilities than detecting SQL injection vulnerabilities due to the imbalance of SQL-SET dataset. The ratio of vulnerable bytecode slices to the invulnerable in the XSS-SET dataset is approximately 0.76:1. However, the ratio in the SQL-SET dataset is 0.11:1. We have also observed that detection of mixed types of vulnerabilities is also less effective than XSS vulnerabilities detection. The precision and F1-measure on the MIX-SET dataset are lower than those on the XSS-SET. For mixed-type vulnerability detection, the neural networks trained by MIX-SET may also identify a vulnerability as another type of vulnerability, in addition to possibly erroneously determining whether target program is vulnerable. These all increase the probability of producing errors. Further, we observed that recall rate is very high and the FPR is very low in the detection results of the three datasets. And the code with and without the vulnerability often only differ between one statement and even a few characters, which indicates that neural networks can distinguish very small vulnerability features.

Second, we find that VulHunter and VulDeePecker are better than other approaches or tools. That is to say, the deep learning-based algorithm for vulnerability detection is effective. We observed that RIPS, sonarqube and phpcs-security-audit have very similar results on all three datasets, with higher recall, FPR, and lower F1-measure (sonarqube is slightly superior to other two). VulHunter and VulDeePecker have more than 80% F1-measure, and both FPR and FNR are below 10%. The highest F1-measure of the other three tools is only 60.21%, but the highest FPR is 100%. Their detection relies mainly on rules

Table 3. Results of comparing with other approaches or tools

Approaches or tools	P(%)	R(%)	F1(%)	FPR(%)	FNR(%)
SQL-SET					
RIPS	13.60	95.65	23.16	69.88	4.35
Sonarqube	12.98	40.22	19.62	31.00	59.78
Phpcs-security-audit	10.60	97.83	19.13	94.88	2.17
VulDeePecker	**86.02**	86.96	86.86	**1.63**	13.04
VulHunter	78.63	**100.0**	**88.04**	3.13	**0.00**
XSS-SET					
RIPS	43.51	97.71	60.21	96.51	2.29
Sonarqube	41.79	43.81	42.78	46.42	56.19
Phpcs-security-audit	42.06	95.41	58.39	100.0	4.59
VulDeePecker	90.04	**97.25**	93.51	8.19	**2.75**
VulHunter	**99.02**	92.22	**95.50**	**0.70**	7.78
MIX-SET					
RIPS	31.42	75.78	44.41	63.58	24.22
Sonarqube	24.44	62.12	35.08	73.85	37.88
Phpcs-security-audit	29.89	94.70	45.43	85.43	5.30
VulDeePecker	76.43	95.47	84.89	8.19	4.53
VulHunter	**85.76**	**97.99**	**91.44**	**6.26**	**2.01**

defined by human experts, and FPs are easily generated when the code of the vulnerability is not very different from the code without the vulnerability.

Third, we observed that VulHunter is better than VulDeePecker in most cases. VulHunter has the highest precision (97.67% in XSS-SET and 85.76% in MIX-SET) and F1-measure (95.13% in SQL-SET, 96.77% in XSS-SET, and 91.44% in MIX-SET). This result can be explained from three aspects.

- We transform the source code into bytecode slices and then use vectors at the bytecode level to represent the vulnerability features. A lot of redundant information, such as comments, custom variable names, is automatically processed after the source code is converted to bytecode, thereby avoiding the over-fitting problem caused by different writing styles.
- After converting the program to bytecode we can see many implicit functions that are not in the source code. This means that we can find more exploit execution paths from bytecode and extract more features, so it can avoid many false negatives.
- We determine whether there is a vulnerability based on the similarity between the target program and vulnerability templates calculated by the neural networks. Templates can preserve the features of the vulnerability so that the neural networks only focus on the calculation of similarity. The template-based similarity calculation method can flexibly cope with variants of different vulnerabilities, and it only needs to add corresponding templates when encountering new vulnerability types.

Using VulHunter in Practice. To further demonstrate the use of VulHunter, we used VulHunter to detect three software developed by PHP: SEACMS, ZZCMS, and CMS Made Simple (CMSMS). These software versions are recently released. According to the data processing steps, we extracted 681, 2,351, and 2,495 suspicious exploit points from three software and then generated bytecode slices. Subsequently, We use VulHunter to detect vulnerabilities and manually verified the results to determine that five of the vulnerabilities were real. Table 4 shows the details of the vulnerability detection results. CVE-2018-19350, CVE-2018-19349, and CVE-2018-20464 are vulnerabilities that have not been published before. They were first discovered and marked in bold in the table. CVE-2018-14962 and CVE-2018-5963 are also real vulnerabilities but have been published by other security researchers.

Table 4. Vulnerabilities detected in real software

CVE-ID	Type	Software	Version
CVE-2018-19350	XSS	SEACMS	6.6.4
CVE-2018-19349	SQL Injection	SEACMS	6.6.4
CVE-2018-14962	XSS	ZZCMS	8.3
CVE-2018-5963	XSS	CMSMS	2.2.8
CVE-2018-20464	XSS	CMSMS	2.2.8

5 Related Work

Vulnerability detection technology is divided into three types: static analysis, dynamic analysis, and mixed analysis [27]. Table 5 shows recent technologies on vulnerability detection. Static analysis is divided into two types: graph-based static analysis and static analysis with data modeling. These technologies has been used by other researchers. For example, Song et al. proposed a method called BitBlaze, which has a static analysis component that can detect vulnerabilities using CFG, DFG and weakest precondition calculation, called Vine [31]. Yamaguchiet et al. proposed a novel and comprehensive representation of source code called *code property graph* that merges concepts of abstract syntax trees, classic program analysis, program dependence graphs, and control flow graphs, as well as help user to model templates for common vulnerabilities with graph traversals [33]. Nguyen et al. propose an enhanced form of CFG known as lazy-binding CFG to produce image-based representation. This technology works well for malware detection [29].

Table 5. Summary of recent technologies on vulnerability detection

Methods	Technologies	Advantages	Disadvantages
Static analysis	Graph-based static analysis	High code coverage	Lack of run-time information
	Static analysis with data modeling		
Dynamic analysis	Fuzzing: AFL, AFLFast, AFLGo, etc.	Fast	Low code coverage
	DTA: DTA, TEMU, DTA++, etc.		
Mixed analysis	Concolic: DART, CUTE, Driller	Fast, high code coverage	Path explosion

6 Conclusions and Future Work

In this paper, we present VulHunter, an automated vulnerability detection system based on deep learning and bytecode. With graph-based static analysis methods, VulHunter can find the code related to the vulnerability and then transform it into bytecode slices that represent the vulnerability well. Bytecode slices are transformed to digital vectors as inputs of neural networks. Our neural networks are different from existing algorithms for direct classification, which can determine whether the target program is vulnerable by calculating similarity between target program and vulnerability templates. Experimental results show that VulHunter achieves 88.04%, 95.50% and 91.44% F1-measure for SQL injection, XSS and mixed types vulnerabilities detection respectively, which are the highest value compared with other approaches or tools. We also tried to use VulHunter to detect three real PHP software (i.e., SEACMS, ZZCMS and CMS Made Simple). Note that three of the five vulnerabilities found have not been published before.

For the future work, we will further the research from two aspects. For one thing, we found that existing automated vulnerability detection methods are powerless for complex vulnerabilities. Complex vulnerabilities have very long ROC-Chains (Return Oriented Programming Chain) and are often associated with multiple files. Therefore, it is difficult to analyze the structure and data information of the program. In the future, we will continue to study the application of deep learning in complex vulnerability detection. For another, due to some indeed limitations of the static analysis method itself, the system is difficult to detect for the vulnerabilities that can occur in execution. For example, some overflow vulnerabilities are easier to detect through dynamic fuzzing techniques. Thus, we will further study the application of deep learning in dynamic vulnerability detection, which can detect the vulnerabilities that only occurs in execution.

Acknowledgments. This work was supported by NSFC-General Technology Fundamental Research Joint Fund (No. U1836215), and the National Key R&D Program of China (No. 2016QY03D0605).

A Appendices

A.1 LSTM Networks

Long short term memory networks, usually just called LSTMs, are a special kind of RNN, capable of learning long-term dependencies. LSTMs contain a complex structure called LSTM cells, which are briefly reviewed below and referred to [25] for greater details.

Each LSTM cell uses a forget gate f (i.e., the state flow of the cell), an input gate i (i.e., the input data), and an output gate o (i.e., the output of module) to control the data flow through the neural networks. Figure 7 shows the detailed structure of the LSTM cell.

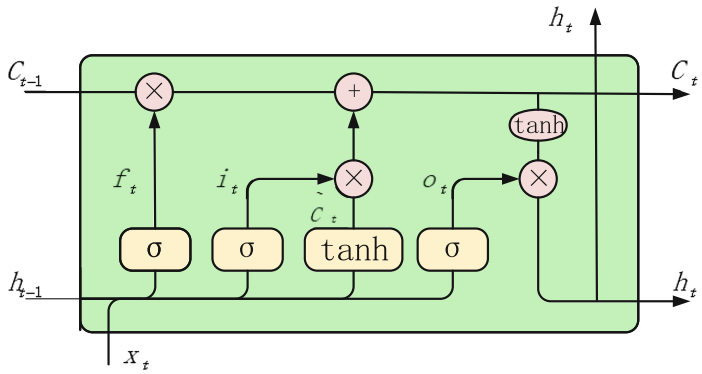

Fig. 7. LSTM cell

The forget gate looks at h_{t-1} and x_t, and outputs a number between 0 and 1, which represents how many percentages of C_{t-1} are retained. The value of f_t at the time t is:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \tag{7}$$

The input gate has two parts. First, a Sigmoid layer outputs i_t that decides which values will update. Next, a *tanh* layer creates a vector of new candidate values, $\widetilde{C}_t$, that could be added to the state. The values of i_t and $\widetilde{C}_t$ at the time t are:

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \tag{8}$$

$$\widetilde{C}_t = tanh(W_c[h_{t-1}, x_t] + b_c) \tag{9}$$

And then the new cell state C_t is:

$$C_t = f_t \odot C_{t-1} + i_t \odot \widetilde{C}_t \tag{10}$$

The output gate is based on Sigmoid layer value o_t and new cell state C_t to calculate the output h_t. The o_t and h_t at the time t are calculated as follows:

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{11}$$

$$h_t = o_t \odot tanh(C_t) \tag{12}$$

where σ denote Sigmoid function $\frac{1}{1+exp(-x)}$, $tanh$ denote the hyperbolic tangent function $\frac{exp(x)-exp(-x)}{exp(x)+exp(-x)}$ and $\odot$ denote the element-wise multiplication, h_{t-1} is output of cell at the time $t-1$, C_{t-1} is state of cell at the time $t-1$, x_t is input of cell at time t, W_f, W_i, W_o, W_C are the weight matrices with the forget gate, the input gate, the output gate, and the cell state input, and b_f, b_i, b_o, b_C are bias items of the forget gate, the input gate, the output gate, and the cell state input.

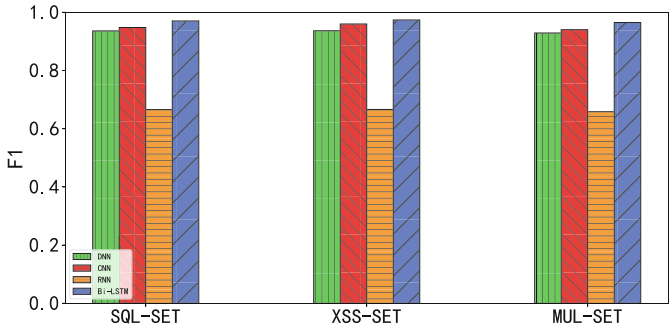

Fig. 8. Neural networks comparison. This figure shows the comparison of DNN (green), CNN (red), RNN (orange), and Bi-LSTM (blue) on the SQL-SET, XSS-SET, and MIX-SET datasets. The ordinate is F1-measures, and the values shown in the figure are the highest values that can be achieved by each network. (Color figure online)

A.2 Neural Networks Comparison

We compared three other types of neural networks: DNN, CNN, and RNN. Similarly, they are all adjusted to determine the best F1-measure. Figure 8 shows the comparison of the best F1-measures for the four neural networks on three datasets. The F1-measures of the four networks are Bi-LSTM, CNN, DNN, and RNN from high to low. The F1-measures of CNN and DNN are not much different at only 0.02 to 0.04 lower than Bi-LSTM. But the value of RNN on the three data sets is only 0.66, which is most likely caused by the vanishing gradients problem (or, exploding gradient problem).

References

1. Flawfinder (2015). http://www.dwheeler.com
2. Acunetix (2018). https://www.acunetix.com
3. Application security statistics report (2018). https://www.whitehatsec.com/blog/2018-whitehat-app-sec-statistics-report/

4. Burpsuit (2018). https://portswigger.net/burp
5. Checkmarx (2018). https://www.checkmarx.com
6. Common weakness enumeration (2018). https://cwe.mitre.org/index.html
7. National vulnerability database (2018). https://www.nist.gov
8. Phpcs-security-audit (2018). https://github.com/FloeDesignTechnologies/phpcs-security-audit
9. RIPS (2018). https://www.ripstech.com
10. Rough audit tool for security (2018). https://code.google.com/archive/p/rough-auditing-tool-for-security
11. Software assurance reference dataset (2018). https://samate.nist.gov/SRD/index.php
12. Sonarqube (2018). https://www.sonarqube.org/
13. Vulcan logic dumper (2018). https://derickrethans.nl/projects.html
14. Word2vec (2018). https://radimrehurek.com/gensim/models/word2vec.html
15. WPScan (2018). https://wpscan.org
16. Bau, J., Bursztein, E., Gupta, D., Mitchell, J.C.: State of the art: automated black-box web application vulnerability testing. In: Security and Privacy, pp. 332–345 (2010)
17. Doupé, A., Cavedon, L., Kruegel, C., Vigna, G.: Enemy of the state: a state-aware black-box vulnerability scanner. In: Usenix Security Symposium (2012)
18. Doupé, A., Cova, M., Vigna, G.: Why johnny can't pentest: an analysis of black-box web vulnerability scanners. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 111–131 (2010)
19. Duchene, F., Groz, R., Rawat, S., Richier, J.L.: XSS vulnerability detection using model inference assisted evolutionary fuzzing. In: IEEE Fifth International Conference on Software Testing, Verification and Validation, pp. 815–817 (2012)
20. Gers, F.A., Schmidhuber, J., Cummins, F.: Learning to forget: continual prediction with LSTM. Neural Comput. **12**(10), 2451–2471 (2000)
21. Gers, F.A., Schraudolph, N.N., Schmidhuber, J.: Learning precise timing with LSTM recurrent networks. J. Mach. Learn. Res. **3**(1), 115–143 (2003)
22. Graves, A.: Long short-term memory. In: Supervised Sequence Labelling with Recurrent Neural Networks. Studies in Computational Intelligence, vol. 385. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-24797-2_4
23. Grieco, G., Grinblat, G.L., Uzal, L., Rawat, S., Feist, J., Mounier, L.: Toward large-scale vulnerability discovery using machine learning. In: ACM Conference on Data and Application Security and Privacy, pp. 85–96 (2016)
24. Halfond W G J, Viegas J, O.A.: A classification of SQL injection attacks and countermeasures (2006)
25. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**, 1735–1780 (1997)
26. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM SIGPLAN Not. **39**(4), 229–243 (2004)
27. Ji, T., Yue, W., Chang, W., Xi, Z., Wang, Z.: The coming era of alphahacking? A survey of automatic software vulnerability detection, exploitation and patching techniques. In: IEEE Third International Conference on Data Science in Cyberspace (2018)
28. Li, Z., et al.: VulDeePecker: a deep learning-based system for vulnerability detection (2018)
29. Nguyen, M.H., Le, N.D., Xuan, M.N., Quan, T.T.: Auto-detection of sophisticated malware using lazy-binding control flow graph and deep learning. Comput. Secur. **76**, 128–155 (2018)

30. Sinha, S., Harrold, M.J., Rothermel, G.: System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In: International Conference on Software Engineering (1999)
31. Song, D., et al.: BitBlaze: a new approach to computer security via binary analysis. In: International Conference on Information Systems Security, pp. 1–25 (2008)
32. William Melicher, A.D.: Riding out DOMSday: Toward detecting and preventing DOM cross-site scripting (2018)
33. Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. In: Security and Privacy, pp. 590–604 (2014)