Vulnerability Detection for Source Code Using Contextual LSTM

Aidong Xu[1,2*], Tao Dai[1,2], Huajun Chen[1,2], Zhe Ming[1,2], Weining Li[3]
1. China Southern Power Grid, Guangzhou, 510080
2. Electric Power Information Security Classified Protection Test and Evaluation Center 5th Lab, Guangzhou, 510080
3. Hainan Power Grid Co.,Ltd, Haikou, 570100
daitworld@126.com

***Abstract*—With the development of Internet technology, software vulnerabilities have become a major threat to current computer security. In this work, we propose the vulnerability detection for source code using Contextual LSTM. Compared with CNN and LSTM, we evaluated the CLSTM on 23185 programs, which are collected from SARD. We extracted the features through the program slicing. Based on the features, we used the natural language processing to analysis programs with source code. The experimental results demonstrate that CLSTM has the best performance for vulnerability detection, reaching the accuracy of 96.711% and the F1 score of 0.96984.**

Keywords-component; vulnerability detection; CLSTM; neural network

I. Introduction

With the rapid development and advancement of Internet technology, computers not only play an increasingly important role in all aspects of our daily lives, but also bring many unavoidable security issues. Software vulnerabilities have become a major threat to current computer security. Software vulnerabilities, refer to phenomena such as abnormal functions, data loss, and abnormal interruptions caused by errors, defects, and weaknesses in the software itself. As a high-level development language, the C/C++ language[1] has high flexibility and code reusability. Due to the defects in the structure of the computer and many current software developments are implemented in the C/C++ language, which makes that the massive security vulnerabilities appeared in the application software process.

According to the statistics of the international authoritative vulnerability release organization on CVE, the number of software vulnerabilities discovered in 1999 was less than 1,600[2]. In 2014, the number of newly discovered software vulnerabilities was close to 10,000. As of now, according to the CVE information database statistics[3], a total of 104,437 software vulnerabilities have been disclosed. In recent years, famous information security incidents have emerged in an endless stream, which caused increasing impact on society. For example, the "Heartbleed"[4] vulnerability exposed in 2014 almost affected the entire Internet, including mail services, instant messaging, online payment, e-commerce and other services, threating the security of Internet users' property and personal information seriously.

Likewise, as a suite of surveillance and spyware for remotely monitoring infected computers, FinSpy (also known as FinFisher) [5] is sold by Gamma International. It is sold to governments and law enforcement agencies around the world. Meanwhile the spyware is used and developed by APT organizations and other hacking organizations. In 2017 BlackOasis (Black Sands), the APT organization, used CVE-2017-8759 in September and FinSpy in CVE-2017-11292 in October[6]. In August 2017, DRIDEX Bank Trojan used CVE-2017-0199 to embed malicious behavior into PPSX files, then they used these files to spread and attack by e-mail [7].

Discovering and eliminating critical vulnerabilities in program code is a key requirement for the secure operation of software systems. There are many research work on vulnerability mining.

In paper[8], Yamaguchi F proposed a source code homology vulnerability detection method based on abstract syntax tree structure pattern in 2012. However, this method can only detect a few simple functional vulnerabilities, and it is difficult to detect complex vulnerabilities across code libraries and cross-functions. In paper[9], He et al. designed a software security vulnerability analysis model based on reverse data correlation in 2013. Based on machine learning, Ren J et al. [10] proposed a software homology vulnerability detection method based on clustering and model analysis. DVCMA (Detecting Software Vulnerabilities Basing on Clustering and Model Analyzing) uses the similarity measurement mechanism based on edit distance and uses the improved k-means algorithm. In paper[11], Hanna S et al. proposed an extensible app analysis framework, Juxtapp, which uses the improved k-gram algorithm to extract code sequence features in 2012. In 2015, F Yamaguchi et al. [12] proposed a detection scheme for Taint-style vulnerabilities, such as "Heartbleed" and "Shellshock". The solution uses a graphical database for vulnerability mining. And the main idea is to build an extended code attribute map. The code attribute map combines three existing program representation methods: abstract syntax tree, control flow graph and program dependency graph. The program can

* Corresponding author.

represent program structure, control flow and data flow. Using the open C / C ++ code analysis platform on the Internet, Joern performs pattern traversal and pattern search on the processed code attribute map contained in the collection to implement vulnerability detection.

Meanwhile, there are some tools for exploiting vulnerabilities. Tools such as IBM AppScan, ASA, and Drozer[13] could mine targeted vulnerabilities, with the higher accuracy and faster. However, this method has obvious defects. It can only detect a small number of simple specific types of vulnerabilities, and could't effectively deal with the changes brought by common program processing methods such as confusion, code optimization, software encryption, etc.

Although previous studies showed the usefulness of prediction models to detection vulnerable components, the improvement of the effectiveness of these techniques is still a challenging problem. In our work, we turned vulnerability identification into a text classification problem, and implement vulnerability detection of source code by combining techniques processed by natural language technology.

Our contributions can be summarized as follows:

1. We introduced the features that using the natural language processing to analysis the program with source code.

2. We collected the massive dataset for C programs with multiple types of vulnerability, including vulnerable and patched.

3. We evaluated the performance on programs based on the Contextual LSTM.

The structure of this paper is as follows: we describe the background information in Section 2. And in Section 3, the system model and algorithm will be elaborated. The results of the experiment and evaluation will be shown in Section 4. Finally, we make a summary of the work and prospects in Section 5.

II. Background Information

A. *Text Classification*

In this paper, we treated the program as a document and used text categorization to identify vulnerabilities. The existence of program vulnerabilities could be understood as missing semantics or confusing errors, such as typical integer overflow vulnerabilities. The integer overflow usually caused by the mixture of unsigned type numbers and signed type numbers as well as the neglect of boundaries during the data operations. For example, solving the 100th item of the fibonacci sequence[14] is no longer possible with ordinary integers. If the developer it not aware of this, it is easy to make such mistakes. The formation of buffer overflow vulnerabilities stems from the fact that the allocation space is too small and the allocation usage restrictions are not strict. It often appears in unsafe string copy functions such as scanf, strcpy, sprintf, and is an abuse of unsafe code modules.

Therefore, we turned vulnerability identification into a text classification problem, and implement vulnerability detection of source code by combining techniques processed by natural language technology.

Common text classification techniques are as follows:

The traditional word bag model, one-hot encoding of words. An N-dimensional zero vector is constructed by indexing the words. If some words in the text appear, the word index value is marked as 1, indicating that the text contains the word. As a way to calculate features weights, TF-IDF is used to evaluate the importance of a word for a document set or a document in a corpus. Meanwhile, there are many word embedding implementation methods, such as Word2Vec[15] and Glove[16].

There are also many neural network models based on deep learning, such as RNN (Recurrent Neural Network)[17], where the current output of a sequence is also related to the previous output. The specific form of expression is that the network memorizes the previous information and applies it to the calculation of the current output. Which means that the nodes between the hidden layers are connected to each other, and the input of the hidden layer includes not only the output of the input layer but also the output of the hidden layers at the previous moment.

However, as time goes on, the hidden layer is multiplied by the weight w again and again. If a weight w is a number close to 0 or greater than 1, the weight value will become very small or large. Which will cause that the number of multiplications increases, with the gradient calculation to become difficult during backpropagation and a gradient explosion. The general RNN model has a poor memory for long-distance information.

LSTM[18] is also known as the long-term and short-term memory network. The LSTM unit consists of the unit status and a bunch of control gates for updating information. The information part is passed to the hidden layer state, so that information memory for long-distance distance can be realized. In our work, we used the Contextual LSTM model[19], an extension of the LSTM, for vulnerability detection.

B. *Source of Dataset*

In our work, We collected datasets from SARD[20], A Software Assurance Reference Dataset， which contains wild, synthetic, and academic security flaws or vulnerabilities. In the SARD, the datasets contain real software application with vulnerabilities and patched. Meanwhile, each program corresponds the certain CWE IDs. It could identify the type of vulnerability that exists in the program.

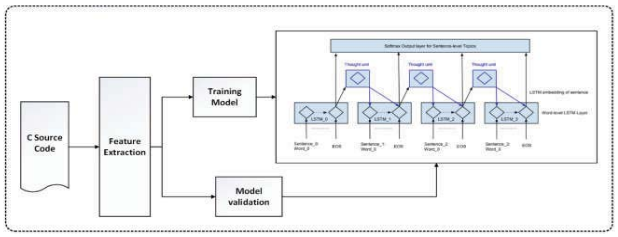


Figure 1. The overall framework

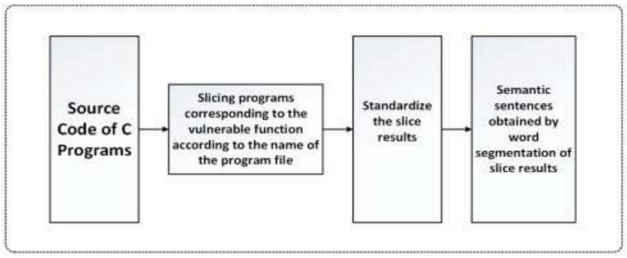

Source Code of C Programs
Slicing programs corresponding to the vulnerable function according to the name of the program file
Standardize the slice results
Semantic sentences obtained by word segmentation of slice results

Figure 2. The process of feature extraction

The datasets contains 23185 programs, including 11093 programs that are vulnerable, and 12092 programs that are not vulnerable. Among the 11093 programs that are vulnerable, 8643 programs correspond to buffer error vulnerabilities (CWE-119) and the rest 2450 programs correspond to resource management error vulnerabilities (CWE-399).

III. System Design

A. Model Overview

In our work, we implemented vulnerability detection based on source code. Through the feature extraction module, we obtained the code features for training the CLSTM neural network, marked the features according to the SARD tags, and used the data for CLSTM learning training, and finally verified the model. The overall framework is shown in the Figure 1.

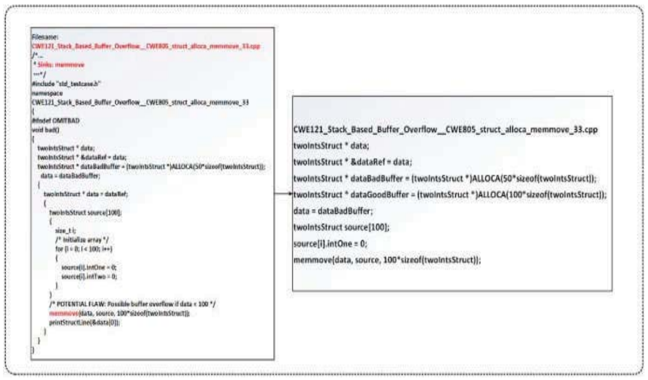
B. Feature Extraction

In this section, We introduced the model of feature extraction used for providing data input for the CLSTM neural network, as shown in the Figure 2. Feature extraction mainly included three steps: program slicing, standardization processing, and word segmentation processing. In our work, we took the C source code as input and treated it as text to identify the vulnerability by text categorization. Therefore, at first, we extracted the semantic text related to the vulnerability by program slicing, corresponding to the vulnerable function according to the name of the program file.

Figure 3 shows the process and result of slicing. We took the CWE-121 vulnerability sample as an example (CWE-121 is a subtype of CWE-119). According to the data set provided by SARD, the name of the vulnerability program is "CWE121_Stack_Based_Buffer_Overflow__CWE805_struct_alloca_memmove_33.cpp". It could be see that "sink" has identified the key function of the vulnerability as memmove in the comments of the source code. Memmove is a function that could be used to copy the values of num Bytes from the location pointed by source to the memory block pointed by destination. Copying takes place as if an intermediate buffer were used, allowing the destination and source to overlap. As shown in the Figure 3, the program contains the library function call memmove, which has three Arguments. We sliced the program according to the passing of the parameters and the call of the function to get the slice sentences. The sentences contained the semantic text related to the vulnerability and were served as input for the subsequent standardization process.

Figure 4 shows the process and result of the standardization process. The input in the example is the output of the Figure 3. The normalization process identified the custom variables and the custom functions, and maped the custom variable to (‘VAR1’,'VAR2','...'). The user-defined functions were identified and mapped to ('FUNC1','FUNC2','...'). Through the standardization process, the semantic influence of the user's naming convention on slice sentences was eliminated, and the result is used as input for the next word segmentation process.

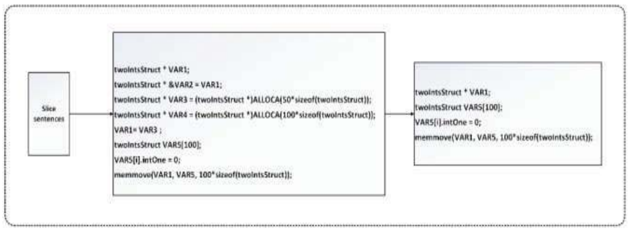
The last step of the word segmentation operation is mainly used to provide a data interface for CLSTM. Through word segmentation, we maped the "twoIntsStruct * VAR1;", obtained in the previous step, to ("twoIntsStruct","*","VAR1",";"). The standardized slice sentences are thus mapped to word vectors and input to CLSTM.



```
Filename:
CWE121_Stack_Based_Buffer_Overflow__CWE805_struct_alloca_memmove_33.cpp
/*...
 * Sinks: memmove
 ...*/
#include "std_testcase.h"
namespace
CWE121_Stack_Based_Buffer_Overflow__CWE805_struct_alloca_memmove_33
{
#ifndef OMITBAD
void bad()
{
  twoIntsStruct * data;
  twoIntsStruct * &dataRef = data;
  twoIntsStruct * dataBadBuffer = (twoIntsStruct *)ALLOCA(50*sizeof(twoIntsStruct));
   data = dataBadBuffer;
  {
    twoIntsStruct * data = dataRef;
    {
      twoIntsStruct source[100];
      {
        size_t i;
        /* initialize array */
        for (i = 0; i < 100; i++)
        {
          source[i].intOne = 0;
          source[i].intTwo = 0;
        }
      }
      /* POTENTIAL FLAW: Possible buffer overflow if data < 100 */
      memmove(data, source, 100*sizeof(twoIntsStruct));
      printStructLine(&data[0]);
    }
  }
}
```

```
CWE121_Stack_Based_Buffer_Overflow__CWE805_struct_alloca_memmove_33.cpp
twoIntsStruct * data;
twoIntsStruct * &dataRef = data;
twoIntsStruct * dataBadBuffer = (twoIntsStruct *)ALLOCA(50*sizeof(twoIntsStruct));
twoIntsStruct * dataGoodBuffer = (twoIntsStruct *)ALLOCA(100*sizeof(twoIntsStruct));
data = dataBadBuffer;
twoIntsStruct source[100];
source[i].intOne = 0;
memmove(data, source, 100*sizeof(twoIntsStruct));
```

Figure 3. The process of slicing



```
twoIntsStruct * VAR1;
twoIntsStruct * &VAR2 = VAR1;
twoIntsStruct * VAR3 = (twoIntsStruct *)ALLOCA(50*sizeof(twoIntsStruct));
twoIntsStruct * VAR4 = (twoIntsStruct *)ALLOCA(100*sizeof(twoIntsStruct));
VAR1= VAR3 ;
twoIntsStruct VAR5[100];
VAR5[i].intOne = 0;
memmove(VAR1, VAR5, 100*sizeof(twoIntsStruct));
```

```
twoIntsStruct * VAR1;
twoIntsStruct VAR5[100];
VAR5[i].intOne = 0;
memmove(VAR1, VAR5, 100*sizeof(twoIntsStruct));
```

Figure 4. The process of standardization

C. *CLSTM For Detection*

In our work, we used the Contextual LSTM as our deep learning model for vulnerability detection. CLSTM (Contextual LSTM), an extension of the recurrent neural network LSTM (Long-Short Term Memory) model, where we incorporate contextual features into the model.

As we know, Convolutional filters can learn good representations automatically, without needing to represent the whole vocabulary. However, CNNs use different window sizes to model the relationship between different scales, leading to lose most of the context information[21].

CLSTM combines CNN and RNN as a classifier for text, and uses the new features obtained by CNN training as input to LSTM. As shown in the Figure 5, in the CLSTM model, the embedding output is not directly access the LSTM layer, but to the CNN layer. This means some sequence is obtained by the CNN, and then these sequences will be access to the LSTM. We can easily add a Conv1D layer and a max pooling layer after the embedding layer, and feeding the consolidated features to the LSTM layer. The hyperparameters of the Conv1D layer and the LSTM layer are the same with the former models. After that, we have 2 dense layers followed and use a sigmoid activation layer to output the result.

A conventional LSTM layer can be expressed as Eq. (1):

$$\begin{aligned}
\boldsymbol{i}_t &= \sigma(\boldsymbol{W}_{xi}\boldsymbol{x}_t + \boldsymbol{W}_{hi}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{ci}\boldsymbol{c}_{t-1} + \boldsymbol{b}_i)\\
\boldsymbol{f}_t &= \sigma(\boldsymbol{W}_{xf}\boldsymbol{x}_t + \boldsymbol{W}_{hf}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{cf}\boldsymbol{c}_{t-1} + \boldsymbol{b}_f)\\
\boldsymbol{o}_t &= \sigma(\boldsymbol{W}_{xo}\boldsymbol{x}_t + \boldsymbol{W}_{ho}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{co}\boldsymbol{c}_{t-1} + \boldsymbol{b}_o)\\
\boldsymbol{c}_t &= \boldsymbol{f}_t \cdot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \cdot \tanh(\boldsymbol{W}_{xc}\boldsymbol{x}_t + \boldsymbol{W}_{hc}\boldsymbol{h}_{t-1} + \boldsymbol{b}_c)\\
\boldsymbol{h}_t &= \boldsymbol{o}_t \cdot \tanh(\boldsymbol{c}_t)
\end{aligned} \tag{1}$$

In Eqn.(1), i,f and o are the input gate, forget gate and output gate respectively, x is the input, b is the bias term, c is the cell memory, and h is the output.

A CLSTM layer is obtained by adding the topic signal T to the input gate. The equation is modified to Eq. (2):

$$\begin{aligned}
\boldsymbol{i}_t &= \sigma(\boldsymbol{W}_{xi}\boldsymbol{x}_t + \boldsymbol{W}_{hi}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{ci}\boldsymbol{c}_{t-1} + \boldsymbol{b}_i + \boldsymbol{W}_{Ti}\boldsymbol{T})\\
\boldsymbol{f}_t &= \sigma(\boldsymbol{W}_{xf}\boldsymbol{x}_t + \boldsymbol{W}_{hf}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{cf}\boldsymbol{c}_{t-1} + \boldsymbol{b}_f + \boldsymbol{W}_{Ti}\boldsymbol{T})\\
\boldsymbol{o}_t &= \sigma(\boldsymbol{W}_{xo}\boldsymbol{x}_t + \boldsymbol{W}_{ho}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{co}\boldsymbol{c}_{t-1} + \boldsymbol{b}_o + \boldsymbol{W}_{Ti}\boldsymbol{T})\\
\boldsymbol{c}_t &= \boldsymbol{f}_t \cdot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \cdot \tanh(\boldsymbol{W}_{xc}\boldsymbol{x}_t + \boldsymbol{W}_{hc}\boldsymbol{h}_{t-1} + \boldsymbol{b}_c + \boldsymbol{W}_{Ti}\boldsymbol{T})\\
\boldsymbol{h}_t &= \boldsymbol{o}_t \cdot \tanh(\boldsymbol{c}_t)
\end{aligned} \tag{2}$$

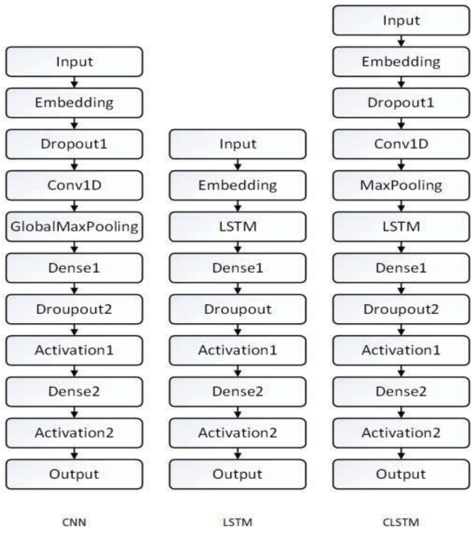


Figure 5. The model of CNN, LSTM and CLSTM

IV. Experiments

A. *Experimental preparation*

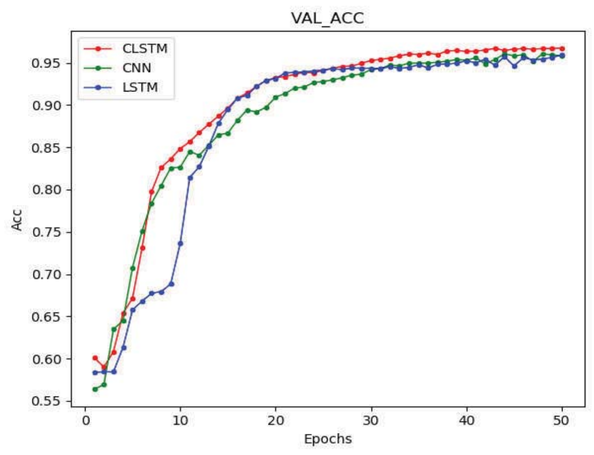
In order to evaluate the performance of CLSTM on vulnerability detection, we implemented the experiment, described in this section. In our work, we implemented the CLSTM neural network in python using Keras as the framework. For training the CLSTM neural network, we adopt the best parameter values corresponding to the effectiveness for vulnerability detection. In the CLSTM model, the dropout is set to 0.5, the batch size is set to 64, the number of epochs is set to 50 and the loss function is adopted to log loss. Table 1 listed the experimental environment for training process.

TABLE I. Experimental environment

Operating system	Ubuntu 18.04 LTS
CPU	Intel (R) Core (TM) i7-7700 CPU @ 3.60GHz
Memory size	20GB

B. *Results*

For evaluating the performance of CLSTM, we compared it with CNN and LSTM on the datasets, which contains 23185 programs, including 11093 programs that are vulnerable, and 12092 programs that are not vulnerable. Figure 6 shows the validation accuracy curves and validation loss curves of CLSTM, CNN and LSTM in 50 epochs. It could be seen that CNN had a higher validation accuracy and



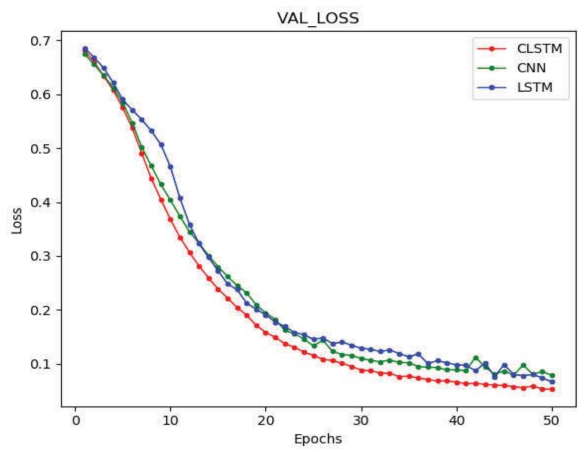


Figure 6. The validation accuracy curves and validation loss curves of CLSTM, CNN and LSTM

increased faster than LSTM before 20 epochs. After 30 rounds, the validation accuracy curves of CNN and LSTM was almost the same. Furthermore, during the 50 epochs, CLSTM has the higher validation accuracy and the lower validation loss than CNN and LSTM, with a higher speed. In the end, CLSTM achieved the validation accuracy of 96.7%, surpassing CNN and LSTM.

To further demonstrate the accuracy and loss of different models, we evaluated the performance on the test dataset according to the following evaluation criteria: Accuracy, Precision, Recall and F1 score. The calculation formula is as Eq. (3):

$$\begin{gathered} TPR = TP/(TP + FN) \\ FPR = FP/(FP + TN) \\ Accuracy = (TP + TN)/(TP + FP + FN + TN) \\ Precision = TP/(TP + FP) \\ Recall = TP/(TP + FN) \\ F = (2 * P * R)/(P + R) \end{gathered} \tag{3}$$

Where TP is False Negative, FP is False Positive, TN is True Negative and TP is True Positive. Table 2 lists the evaluation results of CLSTM、LSTM and CNN on testing dataset. We could see that the CLSTM has the highest F1 score.

TABLE II. EVALUATION RESULTS WITH DIFFERENT MODELS

Model	**Evaluation results**			
	Accuracy	***Precision***	***Recall***	***F1***
CNN	0.95821	0.95897	0.95985	0.95941
LSTM	0.95902	0.96514	0.96089	0.96301
CLSTM	0.96711	0.97068	0.96901	0.96984

C. Discussion

Overall, we have used CLSTM as the classification model to identify vulnerabilities, which have been proved to achieve excellent results, reaching 96.711% accuracy and 0.96984 F1 score. Compared with CNN and LSTM, CLSTM has the best performance on the massive datasets. However, there are still some limitations in our work as follows:

1. Although we used a large number of data sets, the types of vulnerability involved are still insufficient. In our work, we only cover the cwe-119 and cwe-399 vulnerability types. Therefore, in order to train the model more adequately, we need to expand the dataset of multiple vulnerability types.

2. Our model could only be used to detect source code vulnerabilities for C/C++. However, the current programs are not open source, or some are written for Java. Therefore, future work could be studied in programs without source code using deep learning.

V. CONCLUSION

In this paper, we turned vulnerability identification into a text classification problem, and implemented vulnerability detection of source code by combining techniques processed by natural language technology. Through program slicing, we get sentences containing vulnerability semantic information and extract features for training neural networks. In order to ensure the ground truth of the samples, we collected the massive datasets from SARD. Compared with CNN and LSTM, we evaluated the CLSTM on 23185 programs training in 50 epochs. The experimental results demonstrate that CLSTM has the best performance for vulnerability detection, reaching the accuracy of 96.711% and the F1 score of 0.96984. Future work can be developed in expanding the dataset of multiple vulnerability types and parsing programs without source code.

REFERENCES

[1] C/C++ standard library functions, http://en.cppreference.com/.

[2] Martin R A. Managing vulnerabilities in networked systems[J]. Computer, 2001 (11): 32-38.

[3] CVE, http://cve.mitre.org/.

[4] Durumeric Z, Kasten J, Adrian D, et al. The matter of heartbleed[C]//Proceedings of the 2014 Conference on Internet Measurement Conference. ACM, 2014: 475-488.

[5] Marquis-Boire M, Marzcak B, Guarnieri C, et al. You only click twice: FinFisher's global proliferation[J]. 2013.

[6] GReAT. BlackOasis APT and new targeted attacks leveraging zero-day exploit[J]. Securelist., 2017

[7] Read B, Leathery J. CVE-2017-0199 Used as Zero Day to Distribute FINSPY Espionage Malware and LATENTBOT Cyber Crime Malware.Fireeye. com[J]. 2017.

[8] Yamaguchi F, Lottmann M, Rieck K. Generalized vulnerability extrapolation using abstract syntax trees[C]// Proc of Computer Security Applications Conference, 2012:359-368.

[9] He H, Zhao L, Li Q, Zhang W Z, et al. Analyze Software Defects with Program Structure Dependency[C]. In Proceedings of the 2nd International Conference on Computer and Applications, 2013, 17:53-57.

[10] Ren J, Cai B, He H, et al. A Method for Detecting Software Vulnerabilities Based on Clustering and Model Analyzing[J]. 2011.

[11] Hanna S, Huang L, Wu E, et al. Juxtapp: A Scalable System for Detecting Code Reuse among Android Applications[J]. Lecture Notes in Computer Science, 2012, 7591:62-81.

[12] Yamaguchi F, Maier A, Gascon H, et al. Automatic Inference of Search Patterns for Taint-Style Vulnerabilities[C]// Security and Privacy. IEEE, 2015:797-812.

[13] Yang Y, Cai L, Zhang Y. Research on non-authorized privilege escalation detection of android applications[C]// Ieee/acis International Conference on Software Engineering, Artificial Intelligence, NETWORKING and Parallel/distributed Computing. IEEE Computer Society, 2016:563-568.

[14] Horadam A F. A generalized Fibonacci sequence[J]. The American Mathematical Monthly, 1961, 68(5): 455-459.

[15] Goldberg Y, Levy O. word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method[J]. arXiv preprint arXiv:1402.3722, 2014.

[16] Pennington J, Socher R, Manning C. Glove: Global vectors for word representation[C]//Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP). 2014: 1532-1543.

[17] Mandic D P, Chambers J. Recurrent neural networks for prediction: learning algorithms, architectures and stability[M]. John Wiley & Sons, Inc., 2001.

[18] Greff K, Srivastava R K, Koutník J, et al. LSTM: A search space odyssey[J]. IEEE transactions on neural networks and learning systems, 2017, 28(10): 2222-2232.

[19] Ghosh S, Vinyals O, Strope B, et al. Contextual lstm (clstm) models for large scale nlp tasks[J]. arXiv preprint arXiv:1602.06291, 2016.

[20] Black P E. SARD: Thousands of reference programs for software assurance[J]. Journal of Cyber Security and Information Systems-Tools & Testing Techniques for Assured Software-DoD Software Assurance Community of Practice, 2017, 2: 5.

[21] Lai S, Xu L, Liu K, et al. Recurrent Convolutional Neural Networks for Text Classification[C]//AAAI. 2015, 333: 2267-2273.