2023 26th International Conference on Computer Supported Cooperative Work in Design (CSCWD) | 979-8-3503-3168-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/CSCWD57460.2023.10152764

VDDA: An Effective Software Vulnerability Detection Model Based on Deep Learning and Attention Mechanism

Jiaqi Chang[1], Zhujuan Ma[2], Binghao Cao[1], Erzhou Zhu[1*]
[1]School of Computer Science and Technology, Anhui University, Hefei, China
[2]School of Big Data and Artificial Intelligence, Anhui Xinhua University, Hefei, China
[1]cjqi1998@163.com, e20301188@stu.ahu.edu.cn, ezzhu@ahu.edu.cn; [2]zjmsjtu16@163.com

***Abstract*—Many of software vulnerability detection methods suffer from problems of dependent on expert experience, rough detection granularity, and incomplete syntax and semantics information on source codes. This paper proposes the VDDA, Vulnerability Detection based on Deep learning and Attention mechanism, an effective software vulnerability detection model based on deep learning and the attention mechanism. In the VDDA, deep learning technology is used to construct the underlying classifier to avoid the feature engineering of traditional machine learning techniques. The Joren slice tool combined with the code attribute graph (CPG) optimization is used to simplify the source code before it is fed to the Bidirectional Long Short-Term Memory (BLSTM) deep model. Meanwhile, the attention mechanism is employed to improve the efficiency and accuracy of vulnerability detection. Experiment results have demonstrated that the proposed VDDA model is more effective than the existing vulnerability detection methods.**

Keywords—Vulnerability Detection, Software security, Program slicing, Deep learning, Attention mechanism

I. INTRODUCTION

Due to exposure of software vulnerabilities, computer systems are suffering from attacks like Buffer Overflow, Denial of Service (DoS), XSSCross Site Scripting (XSS), and so on. The latest CVE report shows that the number of vulnerabilities collected in the first two quarters of 2022 increased by 31.4% compared with the same period in 2021 [1]. The research on exploiting software flaws or vulnerabilities before or after they are arising is an important topic in the field of program security. Intruders usually deploy their malicious actions by exploiting software flaws. At present, many static and dynamic program analysis methods, such as taint analysis, symbolic execution, dynamic testing, and formal verification, have been widely used in detecting program vulnerabilities [2].

Due to the strong learning ability and the advantage of extracting vulnerability features without expert knowledge, deep learning methods are widely studied and used in software vulnerability detection [3]. Meanwhile, the software vulnerability detection methods that are based on the deep learning frameworks are more accurate and efficient than the traditional machine learning frameworks [4].

The deep learning vulnerability detection frameworks generally convert the source code into vectors (such as code measurements, token sequences, AST, and graphs) that can accurately express the characteristics of the vulnerability. Many of the deep learning-based vulnerability detection methods are working on the coarse-grained file level [5]. However, the coarse granularity will inevitably affect the accuracy on the software vulnerability detection. The syntax and semantic information are also important factors that have great impact on the performance of software vulnerability detection. But in practical works, the performance of many of the existing deep learning methods are downgraded by the deficiency of the syntax and semantic information [6].

This paper proposes the VDDA, an effective software vulnerability detection model based on deep learning and the attention mechanism. The main contributions of this paper can be summarized as follows:

(1) *An effective vectorization method*. In the VDDA, several improvements, including three-direction code slicing, slice organization with code blocks, and separating function names from variable names in code symbolization, are performed to effectively convert source code into vectors for the deep learning framework.

(2) *A CPG optimization algorithm*. In this paper, the Joern (https://joern.io/) slicing tool which converts the target code into code property graphs (CPG) is used to prune the code not related to vulnerabilities. The CPG contains affluent information as AST, control dependency graph (CDG), data dependency graph (DDG), control flow graph (CFG), and program dependency graph (PDG). However, the CPG is more complex than the other graphs. In this paper, the deep-first traversal-based CPG optimization algorithm is proposed to remove CFG and CDG edges dynamically.

(3) *The deep learning and attention mechanism combined architecture for vulnerability detection.* The Bidirectional Long Short-Term Memory (BLSTM) is used to detect software vulnerabilities. In a code block, the importance of slices on the vulnerability detection are not the same. Meanwhile, the importance of code statements and tokens in a slice are also different. In this paper, the attention mechanism is incorporated into the deep learning framework to improve the performance of vulnerability detection.

This study was supported by the University Natural Science Research Projects of Anhui Province, China (KJ2021A0041, KJ2020A1175) and the Natural Science Foundation of Anhui Province, China (2008085MF188).
*Corresponding author (Erzhou Zhu).

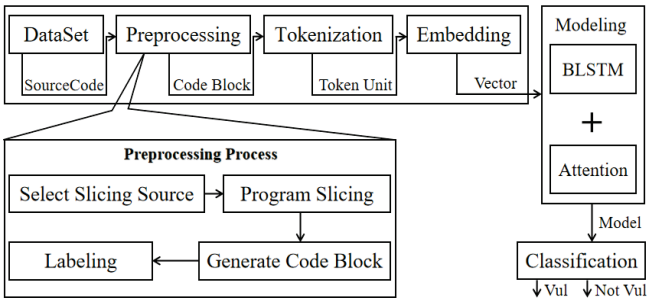


Fig. 1. Workflow of the VDDA.

II. The Proposed VDDA

In the VDDA model, as shown in Fig. 1, the source code is firstly converted into code blocks by sequentially utilizing the program slicing and the CPG optimization. Then, by orderly feeding code blocks to tokenization and embedding modules, the vector representation of the source code is generated. Finally, the neural network integrated with the attention mechanism is trained by the vectors of source code. Workflow of the VDDA.

A. Data Preprocessing

Under normal circumstances, the code related to the vulnerabilities only occupies a small part of the whole program. It is needed to simplify the target code before sending it to the underlying deep learning models. In this paper, the code slicing is used to preprocess the target code to detect vulnerabilities with a finer and more flexible granularity. Specifically, the Joern which converts the target code into CPG is used as the slicing tool of our proposed VDDA model. The CPG contains affluent formation as AST, CDG, DDG, CFG, and PDG. Meanwhile, by the CPG, the syntax, semantics, and other information on the target code can be properly preserved.

A) Slicing source determination

Different selections on the slicing sources usually generate different slicing results of the source code. Since most software vulnerabilities are closely related to API function calls and arithmetic expression (AE), in this paper, source codes of the API and AE are selected as the slicing sources. Specifically, any token nodes in the generated AST fallen into the categories of API and AE can be selected as the slicing sources. For example, the *strncpy* in the sample code in Fig. 2 is an API function where the buffer overflow vulnerability may be occurred. So, the variable "*data*" in Line 23, denoted by <23, data>, can be taken as a slicing source. In addition to API and AE, comparing the vulnerability code with the repaired code, any difference between them can also be taken as the slicing source.

B) Code slicing

After selecting the slicing sources, the CPGs of functions of the entire source code are generated. In the CPG, information on data dependence and control dependence in the

Sample Code 1

```
1  void printLine(const char * line) {
2     if(line!=NULL)
3     {
4        printf("%s\n",line);
5     }
6  }
7  void func_name()
8  {
9     char *data;
10    char *dataBadBuffer= (char*)
          ALLOCA(50*sizeof(char));
11    char *dataGoodBuffer=(char*)
          ALLOCA(100*sizeof(char));
12    if(GLOBAL_CONST_FIVE==5)
13    {
14       data = dataBadBuffer;
15       data[0] = '\0';
16    }
17    {
18       char source[100];
19       memset(source, 'C', 100-1);
20       source[100-1] = '\0';
21       strncpy(data, source, 100-1);
22       data[100-1] = '\0';
23       printLine(data);
24    }
25 }
```

Fig. 2. A sample code for slicing source determination.

target code can be specified by the dependencies among different nodes. The code slicing is performed according to the type of cutting points.

For the function call and AE cutting points, the code that causes vulnerabilities are generally executed before the function call or the arithmetic operations being executed, the backward slicing is performed. For vulnerability code repair or pointer declaration cutting points, the forward slicing is selected. For the composite and the other type of cutting points, the backward and forward slicing are performed simultaneously for retaining more syntactic and semantic information in the code slices.

Sample Code 2

```
1  void slice_test(){
2        int x,y,z,j;
3        x = 1;
4        y = 2;
5        if(x>0){
6           z = y+1;
7        }
8        else{
9           z =4;
10       }
11       if(z < 4){
12          j = 5;
13       }else{
14          j= 6;
15       }
16 }
```

Slice of Sample Code 2

```
1  void slice_test()
2  x = 1;
3  y = 2;
4  if(x>0)
5  z = y+1;
6  z =4;
7  if(z < 4)
8  j = 5;
```

Fig. 3. Sample Code 2 and the corresponding slicing

The slicing process starts from the slicing sources. It cuts off all sentences irrelevant to the slicing sources and retains the sentences that have control dependence or data dependence to the slicing sources. As an example, sentence "*j=5*;" in Line 12 of Fig. 3 is selected as the slicing source. The backward slicing is performed on the *slice_test()*. The CPG of the optimized code and the slicing result are shown in Fig. 3 and Fig. 4 respectively.

C) Code block generation

Different from some existing vulnerability detection methods that directly tokenize the slices, in this paper, the generated slices are further processed to form the informative code blocks. In order to ensure the integrity of the vulnerable

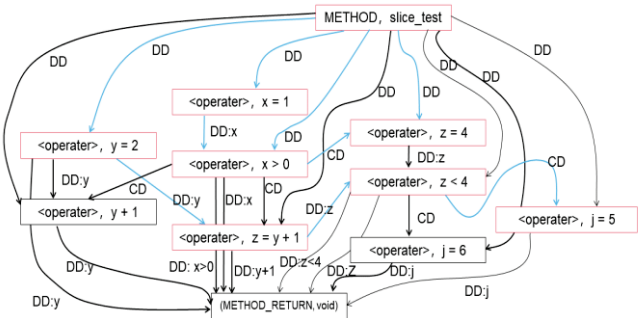


Fig. 4. CPG of the Sample Code 2.

code, in the formation of code blocks, duplicate slices and slices from different directions are removed and merged respectively.

Different slicing directions will make different positions of the useful information in the slice. However, the position of useful information will have great impact on the later stages of our vulnerability detection model. It is needed to assemble code blocks according to the directions of slicing operations.

In this paper, slices generated by the same slicing source are collected at first. Then, according to the single slicing source, the duplicate slices are removed. Finally, code blocks are assembled according to the directions of slicing operations. If the single slicing direction is performed, the slices before or after the slicing source are assembled into a code block. However, if both forward and backward directions are performed, two kinds (forward and backward) of slices are assembled at the tangent point and the code block is formed by removing the duplicate slices in both directions.

D) Code labeling

Code blocks are to be assigned with labels before they can be used for machine learning. By assigning vulnerability code with 1 and normal code with 0, the target code is labeled. In the datasets NVD (https://nvd.nist.gov/) and SARD (https://samate.nist.gov/SRD/index.php/) used in this paper, there are corresponding keywords in the file name to indicate whether the code in this file contains vulnerabilities. For example, in the NVD, the title of the code file is labeled with "VULN" if it contains vulnerabilities. In the SARD, the corresponding label is "BAD".

E) CPG optimization

The Joern slicing tool converts the object functions into CPGs. The CPG is a graph that contains comprehensive information on the object function, including AST, CDG, DDG, CFG, and PDG. By the CPGs, the syntax, semantics, and other information of the target code can be properly preserved. However, the CPG is more complex than the other graphs. For example, the CPG shown in Fig. 4 is quite complicated even for the simple Sample code 2. It is needed to optimize the CPG before the resulting CPG is vectorized.

Generally, the CPG of an object function contains the edges of CFG and CDG. Both edges specify control dependencies in the source code. The redundancy information on control dependencies will degrade the performance of the vulnerability detection. In order to improve the detection efficiency without reducing the vulnerability detection accuracy, a CPG optimization method that is based on the deep-first traversal is proposed to dynamically remove CFG and CDG edges.

Assuming that $F = \{f_1, f_2, \ldots, f_n\}$ is the target program and f_i is a subfunction of program F, the set of CPGs for the subfunctions of F resulting from the Joern platform is $G = \{g_1, g_2, \ldots, g_n\}$. Where g_i is the CPG of the function f_i in the target program and n_{ij} is the j^{th} node of g_i. The CPG optimization method is shown in *Algorithm* 1.

Algorithm 1 CPG optimization

Input: The set of CPGs obtained after the object program F is processed with the Joern tool, $G = \{g_1, g_2, \ldots, g_n\}$.

Output: G_1 //Set of compact CPGs after removing CFG edges or CDG edges from CPGs,

```
G1 ←∅;
for Each gi in G do
    for nodes nij, nik in gi do
        if edge<nij,nik> in CDG then
            if <nij,nik> is in CFG then
                Remove <nij,nik> from CFG;
            end if
        end if
        if edge<nij,nik> in CFG then
            if <nij,nik> is in CDG then
                Remove <nij,nik> from CDG;
            end if
        end if
    end for
    gi →G1;
end for
```

Algorithm 1 performs the depth-first traversal on the edges of CPG g_i in G. In the traversal process on the edges of g_i, if a CFG edge connecting nodes n_{ij} and n_{ij} is firstly encountered, this edge is recorded in the CFG. Thereafter, the CDG edge between nodes n_{ij} and n_{ij} is removed during the remainder traversal of the CPG. Similarly, if a CDG edge connecting nodes n_{ij} and n_{ij} is firstly encountered, this edge is recorded in the CDG. Thereafter, the CFG edge between nodes n_{ij} and n_{ij} is removed during the remainder traversal of the CPG.

B. Tokenization

The name of a function and the variable names in the body of this function do not help much on the performance of vulnerability detection. In order to reduce the interference caused by the uncertainty of names in the training process, function names and variable names in the function body are symbolized before code blocks are tokenized. Specifically, all the function names in a single slice are normalized as "#FUNC#"; all the variable names in the body of the corresponding function are normalized as "#VAR+number".

After the symbolization, the token splitting is performed. In this stage, the lexical analysis is used in the process of tokenized. As shown in Fig. 5, the lexical analysis scans the program and splits each statement into the smaller tokens.

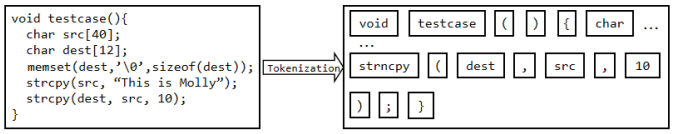


Fig. 5. An example of tokenization.

C. Embedding

It is needed to convert the generated code blocks into vectors to be able to use the deep learning frameworks. Due to the stable performance and the strong anti-interference, in this paper, the FastText is used to vectorize code blocks.

The obtained tokens are firstly formed into a corpus. Then, tokens are taken as the input to the FastText to generate the vector for each token. Finally, by collecting vectors of tokens in a single code block to complete the vectorization. However, due to different code lengths in the code blocks, the number of the split tokens in is different. It is needed to normalize the length of generated vectors.

It is well known that the closer of code to the tangent point, the more effective information it contains. We define the code position close to the tangent point as the expensive end, denoted by *ep*, and the code position far away from the tangent point as the cheap end, denoted by *cp*. As discussing in the previously section, three directions, backward, forward, and backward and forward combined, are incorporated in the process of code slicing. For the forward slicing, the *ep* is at the beginning of the vector and *cp* is at the end of the vector. The *ep* and *cp* of the backward slicing is opposite to that of forward slicing. Take the forward slicing as an example. When the length of a vector is less than the specified value, "0" is padded at the *cp* end. On the contrary, tokens at the *ep* end are pruned to meet the specified length. For bidirectional direction slicing, the *ep* end is the position of the tangent point, and both the starting point and the end point are the *cp* end where "0" can be filled or token can be pruned.

D. Model learning

Since most of the vulnerabilities are not caused by just one code line but are closely related to the context, the BLSTM, is used to detect software vulnerabilities. In a code block, the importance of slices on the vulnerability detection are not the same. Meanwhile, the importance of code statements and tokens in a slice are also different. If all these components are treated equally, the performance of the vulnerability detection will be degraded. In this paper, the attention mechanism is incorporated into the process of deep learning. By the previous embedding stage, the code blocks and the vectorized tokens in these blocks are obtained. The attention mechanism considers the relationships among different code blocks or different tokens in a code block.

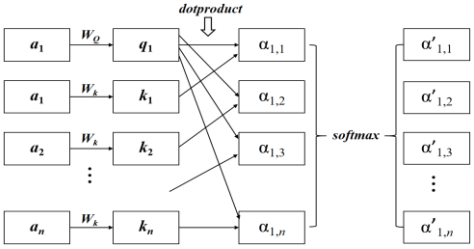


Fig. 6. Formation of the attention score.

In the token level, it is supposed that vectors $\boldsymbol{a}_1$, $\boldsymbol{a}_2$, ..., $\boldsymbol{a}_n$ are generated by tokens $\boldsymbol{token}_1$, $\boldsymbol{token}_2$, ..., $\boldsymbol{token}_n$. As shown in Fig. 6, by orderly multiplying vectors $\boldsymbol{a}_1$, $\boldsymbol{a}_2$, ..., $\boldsymbol{a}_n$ with the query matrix, $\boldsymbol{W}_Q$, and the key matrix, $\boldsymbol{W}_K$, the corresponding query vector $\boldsymbol{q}_1$ and the key vectors $\boldsymbol{k}_1$, $\boldsymbol{k}_2$, ..., $\boldsymbol{k}_n$ are obtained at first. Then, the attention score, denoted by α, between any of two tokens is obtained by performing the dot product operation on the two types of vectors. The attention score specifies the correlation between two tokens, the higher value of α, the stronger correlation between them. Finally, as in equation (1), the value of α is normalized by the *softmax*.

$$\alpha'_{x,y} = \frac{exp(\alpha_{x,y})}{\sum_{j=1}^{j=n} exp(\alpha_{x,j})}, \quad x, y \in [1, n] \tag{1}$$

In equation (1), $\alpha'_{x,y}$ specifies relevance (attention score) between the x^{th} vector and the y^{th} vector in $\{\boldsymbol{a}_1, \boldsymbol{a}_2, ..., \boldsymbol{a}_n\}$.

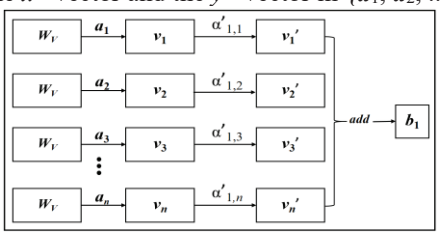


Fig. 7. Formation of the result vector.

In Fig. 7, as in equation (2), the value vectors $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, ..., $\boldsymbol{v}_n$ of the original token vectors are obtained by orderly multiplying vectors $\boldsymbol{a}_1$, $\boldsymbol{a}_2$, ..., $\boldsymbol{a}_n$ with the value matrix $\boldsymbol{W}_V$ (randomly generated).

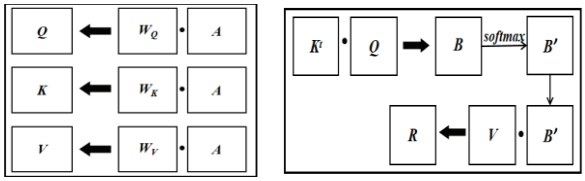
$$\boldsymbol{v}_i = \boldsymbol{W}_V \bullet \boldsymbol{a}_i, \quad i \in [1, n] \tag{2}$$

The result vector $\boldsymbol{b}_x$ is obtained by equation (3).

$$\boldsymbol{b}_x = \sum_{i=1}^{i=n} \alpha'_{x,i} \bullet \boldsymbol{v}_i, \quad x \in [1, n] \tag{3}$$

The method of obtaining attention score in the code block level is the same as the ones in the token level.

Overall, as shown in Fig. 8(a), by separately multiplying the embedding matrix $\boldsymbol{A}$ (composed of token vectors $\boldsymbol{a}_1$, $\boldsymbol{a}_2$, ..., $\boldsymbol{a}_n$) with the query matrix $\boldsymbol{W}_Q$, the key matrix $\boldsymbol{W}_K$, and the randomly generated value matrix $\boldsymbol{W}_V$, three vector spaces, i.e., $\boldsymbol{Q}$, $\boldsymbol{K}$, and $\boldsymbol{V}$, are obtained at first.



(a) Vector spaces (b) Result matrix

Fig. 8. Vector spaces and result matrix generation.

Then, as shown in Fig. 8(b), by performing dot product operation between $\boldsymbol{K}^t$ ($\boldsymbol{K}^t$ is the transpose of matrix $\boldsymbol{K}$) and $\boldsymbol{Q}$ ($\boldsymbol{K}^t \cdot \boldsymbol{Q}$), the attention score matrix $\boldsymbol{B}$ is obtained. The final result vector $\boldsymbol{R}$ is obtained by normalizing attention score matrix $\boldsymbol{B}$ first and multiplying with $\boldsymbol{V}$ second.

The above processes are executed twice. In the first execution, the embedding matrix $\boldsymbol{A}$ is composed of token vectors. In the second execution, the embedding matrix $\boldsymbol{A}$ is composed of code block vectors. By the two executions, the set composed of optimized result vectors is obtained.

III. EXPERIMENTS AND RESULTS

In this part, experiments on determining code block size, embedding method, symbolic algorithm, and the underlying

deep learning model are performed firstly to get the optimal parameters for the proposed VDDA model. Then, effectiveness of the CPG optimization algorithm is tested. Finally, the overall performance of the VDDA model is compared with the ones of five existing models.

The computer that runs these experiments is composed of an AMD 3700x CPU and a NVIDIA RTX 2070 GPU. The dataset used in the experiments are collected from NVD and SARD. Among the 12224 files of this dataset, 2022 files are selected from the open-source project NVD, and 10202 files are selected from the buffer overflow source code vulnerabilities dataset SARD.

A. Code block size and embedding method determination

Different code block sizes and word embedding methods have different impacts on the performance of vulnerability detection. In this experiment, the sizes code blocks vary from 50 to 150 that accompany with three commonly used embedding methods, the Doc2Vec, Word2Vec, and FastText, are verified. As the harmonic average of *Precision* and *Recall*, the performance evaluated by the F_1-*Score* is better than the other indexes.

As the experimental results shown in Fig. 9, all the three embedding methods get the best performance when the code block size reaches 90. Among the three methods, the FastText gets the best F_1-*Score*.

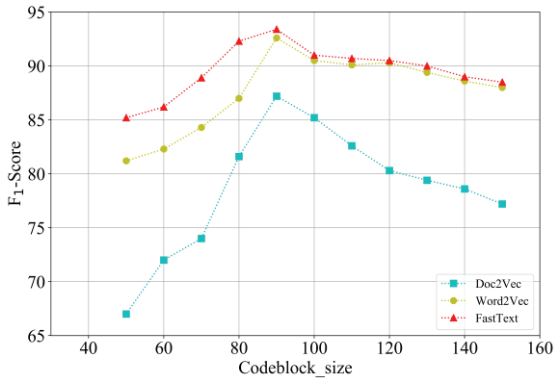


Fig. 9. F_1-*Score* of Doc2Vec, Word2Vec, and FastText with different code block sizes.

B. Symbolization method determination

Before code blocks are tokenized, we need to symbolize the function name and variable names in the function body. In this paper, function names and variable names are treated separately (we call this method as *Type symbolization*). Specifically, all the function names in a single slice are normalized as "#FUNC#"; all the variables in the body of the corresponding function are normalized as "#VAR+number". Meanwhile, the types of data are also recorded.

In order to evaluate the *Type symbolization*, we compare the performance of this method with the ones of the commonly used symbolization method, i.e., the *Single symbolization*. In the *Single symbolization*, the user defined contents, such as function names and variable names, are treated uniformly.

In this experiment, the size of code block is set as 90 and LSTM is selected as the underlying neural network. Meanwhile, performances of different symbolization methods accompany with three commonly used embedding methods, i.e., Doc2Vec, Word2Vec, and FastText, are all tested. The F_1-*Score* is used to evaluate the performance of different symbolization methods.

The experimental results are shown in Fig. 10. In this figure, the "*None*" refers to no symbolic method is used. As the results shown in Fig. 10, all the three embedding methods get the best performance on the *Type symbolization*. Which means the corpus generated by the *Type symbolization* performs better than the ones of the other two symbolization methods.

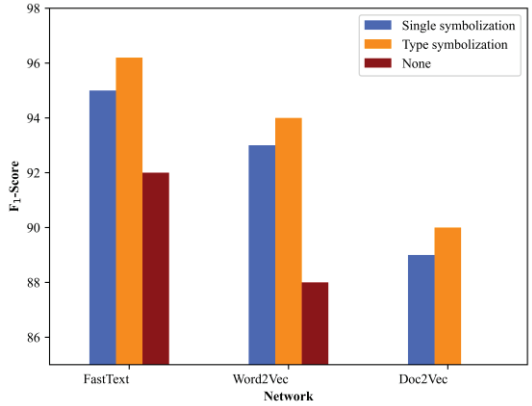


Fig. 10. F_1-*Score* of *Single symbolization*, *Type symbolization*, and no symbolization (*None*).

C. Deep learning model determination

Because most of the vulnerabilities are not only caused by one code line, but are closely related and dependent on the context. Based on this reason, the recurrent neural networks, including BLSTM, Gate Recurrent Unit (GRU), and BGRU, are selected to detect the software vulnerabilities. Meanwhile, the performance of the BLSTM integrated with the attention mechanism (denoted by BLSTM-Attention) is also tested. In the experiments, parameters of the four neural networks are set uniformly.

By setting parameters uniformly, the four neural networks are trained respectively. As the experimental results shown in Fig. 11, the BLSTM-Attention gets the best performance. To this end, the proposed VDDA adopts the BLSTM-Attention to detect software vulnerabilities.

D. Performance of the CPG optimization algorithm

To verify the performance of the new proposed CPG optimization algorithm (*Algorithm* 1), the performance of this algorithm is compared with these of the unprocessed (denoted by "*None*") method, and the method of randomly removing the CFG and CDG edges between any nodes in the CPG (denoted by "*Random*"). Three neural networks, GRU, LSTM, and BLSTM, are used to verify the three CPG processing methods. As the experimental results are listed in Table 1, the *Algorithm* 1 gets the best performance on the four indexes.

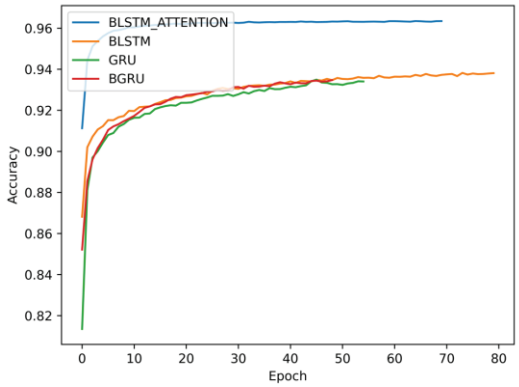


Fig. 11. Accuracy of four different neural networks.

TABLE I. COMPARISONS OF DIFFERENT CPG PROCESSING METHODS.

Algorithm	Networks	*Accuracy*	*Precision*	*Recall*	F_1-*Score*
None	GRU	0.763	0.735	0.835	0.782
	LSTM	0.823	0.794	0.884	0.837
	BLSTM	0.847	0.813	0.891	0.850
Random	GRU	0.673	0.653	0.712	0.681
	LSTM	0.763	0.716	0.803	0.757
	BLSTM	0.758	0.725	0.811	0.766
Algorithm 1	GRU	0.875	0.861	0.902	0.881
	LSTM	0.921	0.916	0.939	0.927
	BLSTM	0.933	0.931	0.941	0.936

E. Overall performance of VDDA

Table 2 lists the experimental results of the VDDA and the five existing software vulnerability detection tools and models. The FlawFinder (http://www.dwheeler.com/flawfinder) and RATS (https://code.google.com/archive/p/rough-auditing-tool-for-security/) are the two commonly used and publicly available software vulnerability detection tools; SySeVR, FUNDED [7], and VulDetector [8] are the representative software vulnerability detection methods proposed recently.

TABLE II. PERFORMANCE COMPARISONS BETWEEN VDDA AND OTHER FIVE EXISTING VULNERABILITY DETECTION METHODS.

Models	*Accuracy*	*Precision*	*Recall*	F_1-*Score*
FlawFinder	0.654	0.490	0.516	0.503
RATS	0.706	0.569	0.538	0.553
SySeVR	0.852	0.772	0.797	0.784
FUNDED	0.894	0.833	0.861	0.847
VulDetector	0.900	0.832	0.884	0.857
VDDA	0.971	0.975	0.952	0.963

From Table 2, the performance of SySeVR is significantly better than these of the FlawFinder and RATS. However, its performance is lower than these of the FUNDED, VulDetector, and VDDA, which convert source code into graphs. The SySeVR framework takes the source code as a token sequence which will lose the syntax and semantic information after token sequence converting vectors.

The VDDA proposed in this paper pays more attention on the processing of CPG. As can be seen from the experimental results listed in Table 2, the performance of VDDA is better than these of FUNDED and VulDetector.

IV. CONCLUSION AND FUTURE WORK

This paper proposed VDDA, an effective software vulnerability detection model based on the deep learning and attention mechanism. In this model, the deep learning technology was used to construct the underlying classifier to avoid the feature engineering of traditional machine learning techniques. The Joren slice tool combined with the CPG optimization was used to simplify the source code before it being fed to the BLSTM deep model. Meanwhile, a variety of methods, such as three-direction code slicing, slice organization with code blocks, and separating function names from variable names in code symbolization, were employed to improve the efficiency and accuracy of vulnerability detection. Experiment results had demonstrated that the proposed VDDA model is more effective than the existing vulnerability detection methods. In the future, studies on expanding and balancing of datasets will be conducted to further improve the performance on the vulnerability detection.

REFERENCES

[1] Common Vulnerabilities & Exposures. Published CVE Records. [Online] Available: https://www.cve.org/About/Metrics.

[2] Quanchen Zou, Tao Zhang, Runpu Wu, et al. From automation to intelligence: Survey of research on vulnerability discovery techniques. Journal of Tsinghua University (Science and Technology), 2018, 58:1079-1094.

[3] Guanjun Lin, Sheng Wen, Qing-Long Han, et al. Software Vulnerability Detection Using Deep Neural Networks: A Survey. Proc. IEEE, 2020, 108(10):1825-1848.

[4] Erzhou Zhu, Qixiang Yuan, Zhile Chen, et al. CCBLA: a Lightweight Phishing Detection Model Based on CNN, BiLSTM, and Attention Mechanism. Cognitive Computation, 2022, [Online] Available: https://doi.org/10.1007/s12559-022-10024-4.

[5] Shouguo Yang, Long Cheng, Yicheng Zeng, et al. Asteria: Deep Learning-based AST-Encoding for Cross-platform Binary Code Similarity Detection. In: Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2021), 21-24 June, 2021, Taipei, pp.224-236.

[6] Zhen Li, Deqing Zou, Shouhuai Xu, et al. SySeVR: a framework for using deep learning to detect software vulnerability. IEEE Transactions on Dependable and Secure Computing, 2021, 19:2244-2258.

[7] Huanting Wang, Guixin Ye, Zhanyong Tang, et al. Combining graph-based learning with automated data collection for code vulnerability detection. IEEE Transactions on Information Forensics and Security, 2021, 16:1943-1958.

[8] Lei Cui, Zhiyu Hao, Yang Jiao, et al. VulDetector: detecting vulnerabilities using weighted feature graph comparison. IEEE Transactions on Information Forensics and Security, 2021, 16:2004-2017