UDC 004

Deep learning based automatic software defects detection framework

A. Chernousov[2,3,a], A. Savchenko[2,3,b], S. Osadchyi[3], Y. Kubiuk[1,3], Y. Kostenko[3], D. Likhomanov[3]

[1] *National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute»*

[2] *National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute», Institute of Physics and Technology*

[3] *Samsung R&D Institute Ukraine (SRK)*

Abstract

We present the VulDetect, a source code vulnerability detection system. This system uses deep learning methods to organizate rules for deciding whether a code fragment is vulnerable. This approach is an improvement of the approach proposed in VulDeePecker. The model uses the AST representation of the source code. We compared vulnerability detection results of both systems on the Bitcoin Core project.

Keywords: vulnerability detection, software vulnerability, analyzer, deep learning, BLSTM, AST

1. Introduction

Various approaches are being intensively developed that help prevents the appearance of software vulnerabilities. Big companies apply the Microsoft SDL [1] procedure for security risks mitigation. SDL recommends specific checks at different stages of the software development life cycle. The most promising is to conduct checks at the stage of implementation of the software. In this paper, we propose an approach for the detection of software defects that can lead to the vulnerabilities, based on deep learning.

The best-known methods for source code defects detection are static analyzers. The main disadvantage of static analyzers is that for each defect expert must write a rule or a set of rules that will detect these defects in the code. At the same time, there are databases of source code, which contain both examples of code with defects, and already corrected. Based on this data, one can take advantage of deep learning to automatically build rules for source code analysis. The primary challenge of deep learning based approach is the construction of informative code representation, that will take into account both functionality and context of code chunks. Contributions of this paper are to improve the technology of extraction code fragments and the representation of the source code. [2].

2. Related Work

There currently exists a wide range of utilities for finding defects in the program code. Analyzers work with a different representation of the source code: a pure source code, an Abstract Syntax Tree (AST) and executable binary files [3] [4]. From the entire list of utilities, you can define systems of static analyzers with open source [5] [6] [7] [8] [9], commercial products [10] [11] [12] [13], and some research projects [4] [14] [15] [16] [17] [18] [19]. Based on analysis approach, static analyzers can be divided into 2 types:

1) Rule-based [20]
2) Code similarity-based [21].

2.1. Rule-based approach

This approach has two main problems, which consist of intensive manual labor and monotonous work and high false negative rates.

1) Intensive manual labor and monotonous work
   Here, task of building rules according to which the vulnerabilities of a particular function will be determined (discovered) rely on security specialist. This task is quite tedious and subjective. Sometimes, errors occur due to the complexity of the implemented functions. In other words, to identify the signs of vulnerability, many aspects need to be taken into account. In principle, the solution to this problem consists of the independent writing of the same function by several experts, and then the choice of the most effective, or combinations of specific functions. However, this leads to even more manual labor. There is a tendency to automate cyber defense, which is stimulated by the DARPA's Cyber Grand Challenge [22], so it is desirable to reduce or eliminate the reliance on manual labor, whenever it is possible. Therefore, it is essential to fence people from the tedious and subjective task of manually defining rules for detecting vulnerabilities.
2) High false negative rates

[a] adrerek@gmail.com

[b] artichsa@gmail.com

On the other hand, existing solutions often overlook many vulnerabilities. In other words, have a high level of false negative results. According to the results of article [23], these indicators for the Clang Analyzer [8] or [9] were 84% and 92%, respectively. These values may be justified by the emphasis on a low level of false positive results, but this is still not a very good indicator.

2.2. Code similarity-based approach

The Code similarity-based approach also has two problems: the absence of a data set and absence of an efficient comparison algorithm.

1) Absence of a data set
   At the moment, there are no open datasets suitable for this task. This problem is relevant, even though the National Vulnerability Database (NVD) [24] and the Open Sourced Vulnerability Database (OSVDB) [25] have become open. Also, in articles [26] [27], databases were created for matching numbers and identifiers of Common Vulnerabilities and Exposures (CVE-ID) [28] with commits, where each commit contains the difference of the source code before and after commit. However, all this data is not enough to identify all vulnerabilities.
2) Absence of an efficient algorithm
   There currently no single efficient code similarity algorithm that would be effective for all types of vulnerabilities, since each vulnerability has its characteristics that should be taken into consideration.

An improved approach of Code similarity-based was published in article [2]. The advantage of this method over the Code similarity-based approach is an efficient algorithm for finding similar code with using Deep Learning. Compared to the Rule-based approach, it has a rather low false negative rate, about 7%. The disadvantages of this method are: there is still not exists of a sufficient data set and higher false positive rates.

It can be said that the vulnerability detection system with high a false positive rate may be unsuitable for use, and systems with a high false negative rate may be useless. This justifies the importance of using systems that can provide low false negative rates, while the false positive rates are not too high.

3. Statement of the problem

This paper discusses the problem of building a source code analysis system based on deep learning that allows you to determine whether a code fragment contains a defect (memory leaks, buffer overflow, etc.). The advantages of this approach are the ability to automatically formulate a rule for deciding whether a code fragment is vulnerable or not, based on the accumulated experience of writing code and fixing defects. Deep learning methods have a good ability to extract generalized patterns from a large amount of data, which in this case is an accumulated code base.

The object of study is the software development process.

The subject of the study is the detection of defects in software source code.

Research methods - AST representation of source code, deep learning methods for solving classification problems.

The scientific novelty of this work consists in applying the AST representation of the source code and the principle of its processing to extract a code fragment (code gadget), which made it possible to improve the accuracy of defects detection. This approach is an improvement of the technology described earlier in the article [2].

4. Major principles for deep learning-based vulnerability detection

In this section, we propose to consider some basic principles of using deep learning to detect vulnerabilities. Some principles may need to be improved or studied in more detail, but this is sufficient for current research on the detection of vulnerabilities. When using neural networks, standard basic questions always arise before starting development, and in this research, the following questions arose: 1. How to represent a program that is tested for vulnerability detection system based on deep learning? 2. How to localize a vulnerability? 3. What neural network architecture to choose?

4.1. How to represent program?

Since the neural network accepts numerical vectors at the input, it is necessary to transform the program in such a way as to keep the relation between the vector and the semantic information about the program. In articles [3] [4], there were proposals to work with binary program data and the AST, and with the source code of the program, as well. In other words, we need a method that will create a relation between the program presentation and the vector representation, which is input to deep learning. This led to the following steps:

1) The program is first converted to AST
2) From all, the code gadget is extracted (the part of the code that refers to the call of a certain function or the arguments of the function call)
3) Through word2vec [29] code gadget converted into a vector, which is the input data for the neural network.

4.2. How to localize a vulnerability?

Detection of the vulnerable code is not the only challenge to be solved. An essential task is also finding the location of a vulnerability. This means that vulnerability detection should not be carried out at a program or function level that is too abstract. This led to the following case: In order to determine the more precise location of a vulnerability, a program must be presented with a higher degree of detail than the program or function as a whole. Indeed, the presentation of the code gadget leads to more accurate vulnerability position detection, because, in most cases, the code gadget consists of only a few lines.

4.3. What neural network to choose?

Neural networks have proven themselves in the following areas: image processing, speech recognition, clustering, which differ from finding vulnerabilities. From this follows that many types of neural networks are not suitable for our purposes and that the neural network we need must have specific properties. This led to the following cases: Since a particular line of vulnerability code may depend on the context of the program code, neural networks that can work with the context will suit, an example would be neural networks for processing human speech. One can see that function call arguments are most often influenced by earlier or later operations in the program. We begin our consideration with a neural network with feedback, that is a Recurrent Neural Network (RNN) [30]. But this neural network has one major drawback, namely the vanishing gradient problem [31] [32]. To address this issue, more complex architecture has been chosen, namely, a neural network with a long short-term memory (LSTM) [33]. However, even the LSTM neural network in its standard form does not fit, since the network is unidirectional, and because the arguments of the function can be affected by both earlier and later operations in the program. From this, it follows that a unidirectional LSTM network may not be enough. Therefore, it was decided to use a bidirectional LSTM (BLSTM) [34].

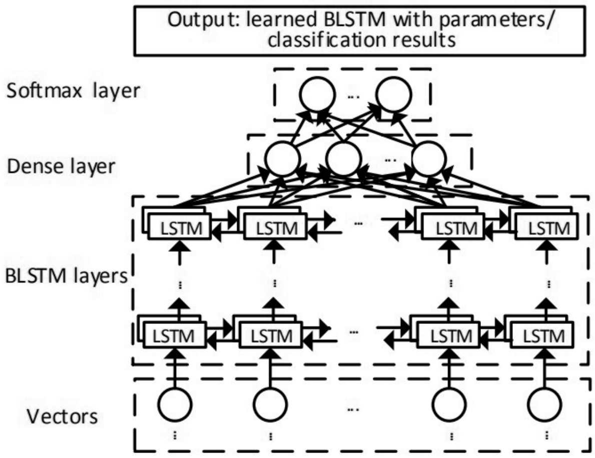

Output: learned BLSTM with parameters/ classification results
Softmax layer
Dense layer
BLSTM layers
LSTM
Vectors

Fig. 1. A brief review of BLSTM neural network

The Figure 1 shows a block diagram of a BLSTM neural network, with several BLSTM layers, a dense layer, and a softmax layer. The entrance to the neural network at the training stage is a specific vector representation. The BLSTM layers themselves have two directions: forward and backward and contain several complex LSTM cells. The dense layer reduces the number of vectors obtained as a result of the BLSTM layer, and the softmax layer accepts vectors from the dense layer as input and is responsible for the presentation and formatting of the classification result, which provides feedback for updating the neural network parameters during the learning phase. The result of the learning phase is the BLSTM neural network with precisely tuned model parameters, and the output of the detection phase is the classification results.

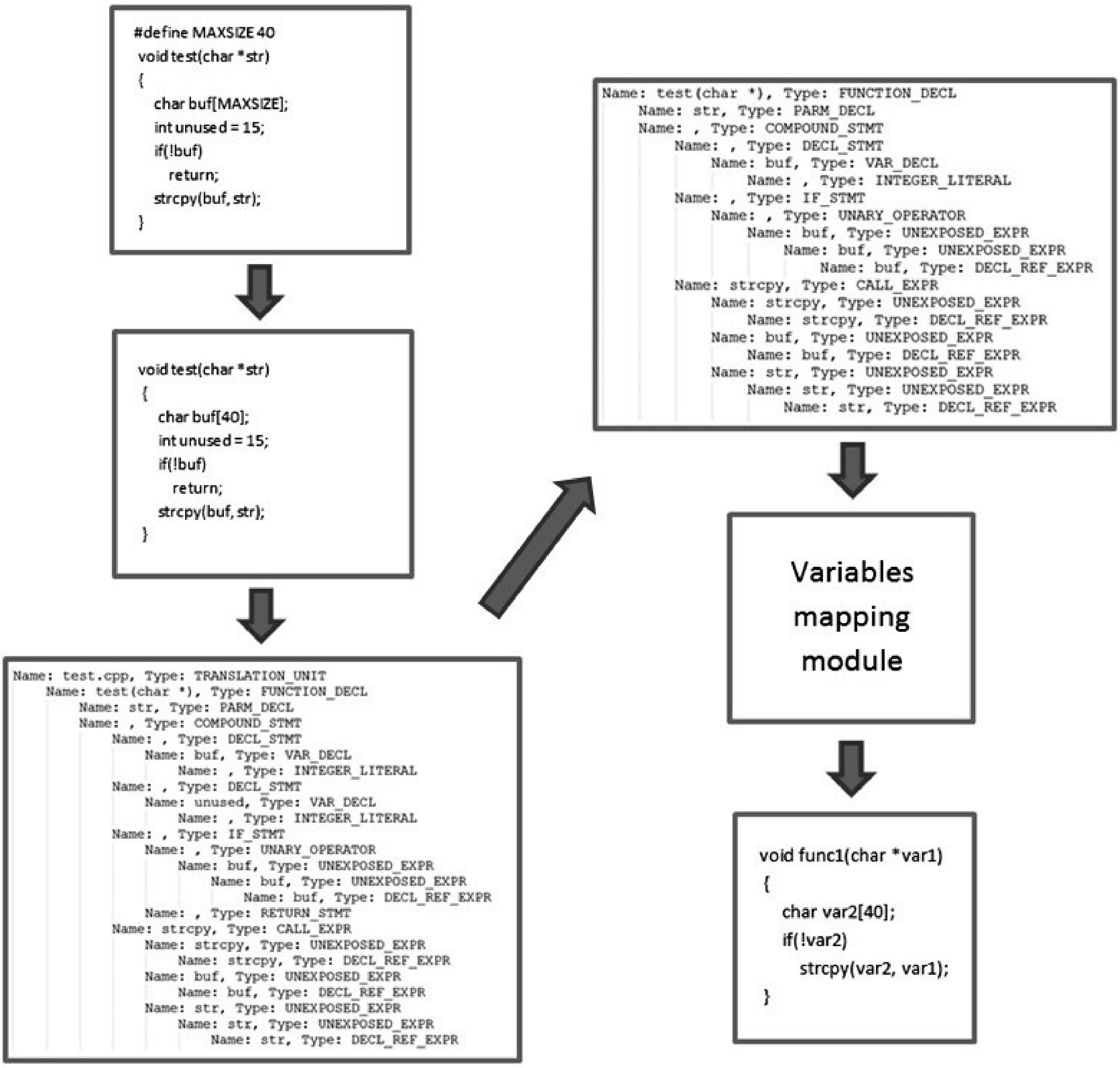
5. Design

Our objective is to design a vulnerability detection system (VulDetect) that can automatically tell whether a given program in the source code is vulnerable or not and if so, the locations of the vulnerabilities. This should be achieved without asking human experts to define features manually and without incurring high false negative rates (as long as the false positive rates are reasonable). In this section, we describe the design of VulDetect. We start with a discussion on the notion of code gadget because it is crucial to the representation of programs.

5.1. Extracting code gadgets

Working with the source code, we need to decide which part of the code we should work with, this can be functions or strings or the entire file. We need to extract from the source code a piece of code that will be associated with a potential vulnerability, and with which we will work in the future. This piece of code will be called Code Gadget by analogy with article [2]. Code gadget is a set of statements influenced by a single data stream (data flow). We will not extract the code gadget from raw text, but in advance transformed into an AST. This is a data structure that represents the source code in the form of a tree, where each node is associated with a language construct that is found in the source code. There are following steps to extract code gadgets demonstrated in Figure 2:

1) Preprocessor. Based on the source code, using the built-in clang preprocessor, a new source code file is created without preprocessor directives that have been defined by a user.
2) Build AST. To build AST, we use the clang compiler toolkit.
3) Search for start points. Under the start point means the place in the source code, which begins the analysis. We use function calls from the standard C library as a start point (for example, malloc, memcpy, fopen, etc.)
4) Building a dependency graph. At this stage, we build a dependency graph of the arguments (or return values) on the start point; in other words, the function. Functions can belong to one of two types: backward and forward. The backward type includes those functions for which it is important to monitor the previous state of the arguments (for example, malloc). The forward type includes those functions for which it is important to monitor the status of the returned values.
5) Obfuscation of user variables or functions. On this step, we get rid of the dependencies of the names of user functions and variables. We replace all names of functions and variables with symbolic names such as: «var1», «var2», etc. and «func1», «func2», etc. At the same time, different names



```
#define MAXSIZE 40
void test(char *str)
{
   char buf[MAXSIZE];
   int unused = 15;
   if(!buf)
      return;
   strcpy(buf, str);
}
```

```
void test(char *str)
{
   char buf[40];
   int unused = 15;
   if(!buf)
      return;
   strcpy(buf, str);
}
```

```
Name: test.cpp, Type: TRANSLATION_UNIT
    Name: test(char *), Type: FUNCTION_DECL
        Name: str, Type: PARM_DECL
        Name: , Type: COMPOUND_STMT
            Name: , Type: DECL_STMT
                Name: buf, Type: VAR_DECL
                    Name: , Type: INTEGER_LITERAL
            Name: , Type: DECL_STMT
                Name: unused, Type: VAR_DECL
                    Name: , Type: INTEGER_LITERAL
            Name: , Type: IF_STMT
                Name: , Type: UNARY_OPERATOR
                    Name: buf, Type: UNEXPOSED_EXPR
                        Name: buf, Type: UNEXPOSED_EXPR
                            Name: buf, Type: DECL_REF_EXPR
                Name: , Type: RETURN_STMT
            Name: strcpy, Type: CALL_EXPR
                Name: strcpy, Type: UNEXPOSED_EXPR
                    Name: strcpy, Type: DECL_REF_EXPR
                Name: buf, Type: UNEXPOSED_EXPR
                    Name: buf, Type: DECL_REF_EXPR
                Name: str, Type: UNEXPOSED_EXPR
                    Name: str, Type: UNEXPOSED_EXPR
                        Name: str, Type: DECL_REF_EXPR
```

```
Name: test(char *), Type: FUNCTION_DECL
    Name: str, Type: PARM_DECL
    Name: , Type: COMPOUND_STMT
        Name: , Type: DECL_STMT
            Name: buf, Type: VAR_DECL
                Name: , Type: INTEGER_LITERAL
        Name: , Type: IF_STMT
            Name: , Type: UNARY_OPERATOR
                Name: buf, Type: UNEXPOSED_EXPR
                    Name: buf, Type: UNEXPOSED_EXPR
                        Name: buf, Type: DECL_REF_EXPR
        Name: strcpy, Type: CALL_EXPR
            Name: strcpy, Type: UNEXPOSED_EXPR
                Name: strcpy, Type: DECL_REF_EXPR
            Name: buf, Type: UNEXPOSED_EXPR
                Name: buf, Type: DECL_REF_EXPR
            Name: str, Type: UNEXPOSED_EXPR
                Name: str, Type: UNEXPOSED_EXPR
                    Name: str, Type: DECL_REF_EXPR
```

```
void func1(char *var1)
{
   char var2[40];
   if(!var2)
      strcpy(var2, var1);
}
```

Fig. 2. Steps to extract code gadget

from different code gadget can be mapped into the same symbolic name.

6) Generate code gadget. Based on the graph, a code gadget is formed from tokens taken from the nodes of the graph.

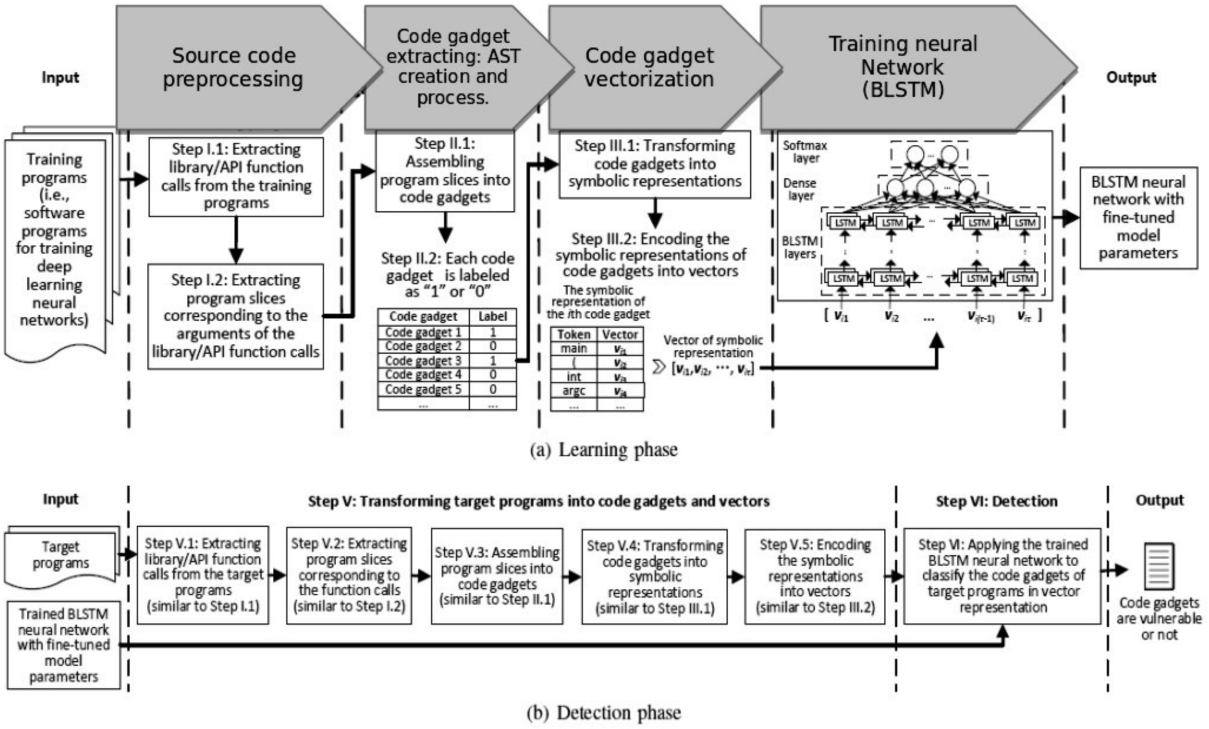
5.2. Approach details

When we extract the code gadget from the context file, it is necessary to take into consideration all dependencies from internal files or libraries and external ones. First of all, we downloaded and saved the Windows OS libraries into a separate folder, and when analyzing the source code files, in addition to the .c/.cpp files, we also found and included .h/.hpp files. This made it possible to view the code in a correct form because sometimes the operators that affect variables were not limited to one file. After that, we encoded the converted code gadget into a numerical vector, whose indexes will correspond to the line number of the code in the current code gadget. To achieve the desired result, we used word2vec [29], which allowed us to preserve the lexical and logical relationships within the vector for each code segment.

We use a supervised learning model, and each input vector was marked as 1 or 0, respectively vulnerable and not. The following structure of the model was chosen, which had n_epochs/batch_size = 100 and consisted of four parts:

1) five BLSTM layers, at the input and output of which was a vector of size N_features;
2) one dense layer of «rele» type had the same size;
3) dense layer of type «softmax», which took a vector of size N_features, and narrowed it to six;
4) the last layer used the loss function «binary_crossentropy» and, as a result, returned a value of type «float» from 0 to 1.

For writing this model, numpy and keras were used.


Input
Source code preprocessing
Code gadget extracting: AST creation and process.
Code gadget vectorization
Training neural Network (BLSTM)
Output
Training programs (i.e., software programs for training deep learning neural networks)
Step I.1: Extracting library/API function calls from the training programs
Step I.2: Extracting program slices corresponding to the arguments of the library/API function calls
Step II.1: Assembling program slices into code gadgets
Step II.2: Each code gadget is labeled as "1" or "0"
Code gadget | Label
Code gadget 1 | 1
Code gadget 2 | 0
Code gadget 3 | 1
Code gadget 4 | 0
Code gadget 5 | 0
... | ...
Step III.1: Transforming code gadgets into symbolic representations
Step III.2: Encoding the symbolic representations of code gadgets into vectors
The symbolic representation of the ith code gadget
Token | Vector
main | v_{i1}
(| v_{i2}
int | v_{i3}
argc | v_{i4}
... | ...
Vector of symbolic representation
$[v_{i1}, v_{i2}, \cdots, v_{i\tau}]$
Softmax layer
Dense layer
BLSTM layers
LSTM
$[v_{i1}$ v_{i2} ... $v_{i(\tau-1)}$ $v_{i\tau}]$
BLSTM neural network with fine-tuned model parameters
Input
Step V: Transforming target programs into code gadgets and vectors
Step VI: Detection
Output
Target programs
Trained BLSTM neural network with fine-tuned model parameters
Step V.1: Extracting library/API function calls from the target programs (similar to Step I.1)
Step V.2: Extracting program slices corresponding to the function calls (similar to Step I.2)
Step V.3: Assembling program slices into code gadgets (similar to Step II.1)
Step V.4: Transforming code gadgets into symbolic representations (similar to Step III.1)
Step V.5: Encoding the symbolic representations into vectors (similar to Step III.2)
Step VI: Applying the trained BLSTM neural network to classify the code gadgets of target programs in vector representation
Code gadgets are vulnerable or not

(a) Learning phase

(b) Detection phase

Fig. 3. General data-flow of vulnerabilities detection based on DL

As a result, the analysis of vulnerabilities of the project, showed the probability of vulnerability of C/C++ files consisted of the following steps that illustrated on Figure 3:

1) extracting C/C++ files and their dependencies;
2) AST conversion and localization of potentially vulnerable parts of the code;
3) transformation of code sections in code gadgets;
4) encoding code gadget into a numerical vector;
5) using the neural model to find vulnerable parts of the source code.

5.3. Approach weaknesses

The proposed approach has the following weaknesses:

1) The source codes from the training sample contained specific Windows and *nix system libraries, so to correctly create the code gadget, we needed a manual installation of the missing dependencies. Otherwise, parts of the code associated with unknown libraries did not fall into the AST, which significantly reduced the size and quality of the training sample.
2) Also, an important aspect of the source code processing is that in order to correctly extract code gadgets, it is necessary to make a list of potentially vulnerable functions manually and set a list of their features, for example, whether this function is forward or backward, whether it is necessary to track function arguments, whether it is necessary to track returned function values.
3) In order for the code gadgets did not include information from third-party files and libraries; it is required to manually set the directory with the files on which the code gadgets will be built. Otherwise, as a result of passing through the nested functions, the code gadget will have information that is not relevant from the user's point of view and potentially may introduce an error during the analysis.
4) When we track the arguments or return values of the function, it rarely occurs cases with a large number of nestings, as a result of which the code gadgets become quite large. This problem entails a number of consequences:
   a) decreases the accuracy of the model results;
   b) from the user's point of view, the informativeness of the resulting report falls because the description of the vulnerable part of the code is too inaccurate;
   c) in general decreases system performance because more information needs to be processed.

   This problem was partially managed to get rid of by manually setting the bound level of nesting. This solution has reduced the size of the code gadgets and increase the accuracy of the analysis, but it can potentially miss important details of the code gadget.

6. Results of work

As part of the research, was made a comparison of two models: VulDeePecker and VulDetect. To train the VulDeePecker model as a training sample, pre-prepared code gadgets were taken from the authors of the original article [2]. For training the VulDetect model, own code gadgets were generated using the algorithms mentioned above. Both training samples were created based on source codes taken from the National Vulnerability Database (NVD) [24], and from the NIST Software Assurance Reference Dataset (SARD) [35]. The initial sample contains 8122 sample code with the presence of a buffer overflow vulnerability, as well as 1729 code samples with vulnerabilities associated with incorrect resource management. The sample was divided into training and test in the ratio of 80% and 20%, respectively. Comparative characteristics of the output accuracy of training and testing for VulDeePecker and VulDetect models are presented in Table 1.

Table 1. Comparison of VulDeePecker and VulDetect in accuracy on different phases

Phase/accuracy	**VulDeePecker**	**VulDetect**
Learning	96%	97%
Testing	91%	92%

As can be seen from the Table 1, the accuracy of the model VulDetect exceeds the accuracy of the model VulDeePecker. This is due to the use of another algorithm for generating gadget code that we proposed.

As an experiment, the trained VulDeePecker and VulDetect models were used to search for vulnerable areas in the source codes of the Bitcoin protocol implementation, available from the [36]. The results are presented in Table 2.

Table 2. Comparison of VulDeePecker and VulDetect in different metrics of number of finded code gadgets

Metrics	**VulDeePecker**	**VulDetect**
Original number of code gadgets	1510	
Number of code gadgets marked as «safe»	970	1326
Number of code gadgets marked as «vulnerable»	540	184
Number of code gadgets marked as «safe» by both models	802	802
Number of code gadgets marked as «vulnerable» by both models	16	16
The number of unique code gadgets marked as «safe»	168	524
The number of unique code gadgets marked as «vulnerable»	524	168

The anomaly rating for each code gadget of the original sample is presented in Figure 4:

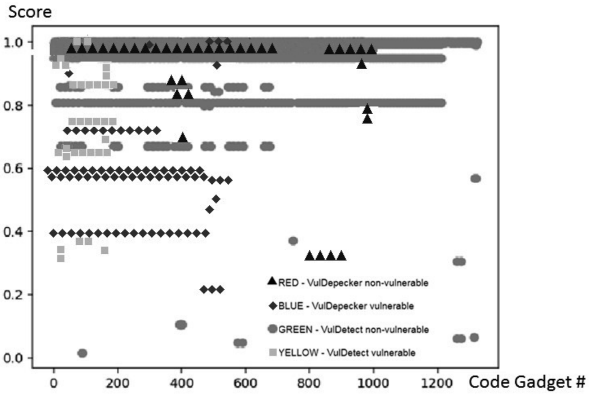


Fig. 4. Comparison of VulDeePecker and VulDetect

According to the results of the experiment, it is clear that the VulDeePecker model marked as "vulnerable" a much larger number of gadgets than the VulDetect model, but with a lower anomaly index, which indicates a higher false positive rate.

7. Conclusion

In this paper presented a new DL-based source code vulnerability detection system called VulDetect, which is an improvement of VulDeePecker technology. The model uses the AST representation of the source code. A new code gadget extraction system has been developed. We compared the results of detecting vulnerabilities of both systems on the Bitcoin Core project. In the VulDetect model, the false positive rates were reduced, which led to increase in false negative rates. There is still a problem with a valid dataset, which will increase the accuracy of neural network predictions and increase the number of detected vulnerabilities. In the future, it is planned to make a more efficient algorithm for extracting the code gadgets, which will increase the accuracy of vulnerability detection in the source code.

References

[1] "Microsoft security development lifecycle (sdl)." `http://www.cs.fsu.edu/~jowett/MS_SDL_Version_3.2.pdf`. Accessed: 2019-03-26.

[2] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," in *Proceedings 2018 Network and Distributed System Security Symposium*, Internet Society, 2018.

[3] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.* (E. Kirda and T. Ristenpart, eds.), pp. 99–116, USENIX Association, 2017.

[4] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proceedings of the 28th Annual*

Computer Security Applications Conference on - ACSAC '12, ACM Press, 2012.

[5] "Flawfinder." `https://dwheeler.com/flawfinder/flawfinder.pdf`. Accessed: 2019-03-26.

[6] "Rough auditing tool for security." `https://code.google.com/archive/p/rough-auditing-tool-for-security/`. Accessed: 2019-03-26.

[7] J. Viega, J. Bloch, Y. Kohno, and G. McGraw, "ITS4: a static vulnerability scanner for c and c++ code," in *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*, IEEE Comput. Soc, 2000.

[8] D. Marjamaki, "Cppcheck - a tool for static c/c++ code analysis." `http://cppcheck.wiki.sourceforge.net/`. Accessed: 2019-03-26.

[9] B. D. Fandrey, "Clang/llvm." `https://llvm.org/pubs/2010-06-06-Clang-LLVM.pdf`. Accessed: 2019-03-26.

[10] "Checkmarx." `https://www.checkmarx.com/`. Accessed: 2019-03-26.

[11] "Coverity." `https://scan.coverity.com/`. Accessed: 2019-03-26.

[12] "Hp fortify." `https://www.hpfod.com/`. Accessed: 2019-03-26.

[13] "Dexter." `https://github.com/Samsung/Dexter`. Accessed: 2019-03-26.

[14] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," in *Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy - CODASPY '16*, ACM Press, 2016.

[15] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: A scalable approach for vulnerable code clone discovery," in *2017 IEEE Symposium on Security and Privacy (SP)*, IEEE, may 2017.

[16] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "VulPecker," in *Proceedings of the 32nd Annual Conference on Computer Security Applications - ACSAC '16*, ACM Press, 2016.

[17] S. Neuhaus and T. Zimmermann, "The beauty and the beast: Vulnerabilities in red hat's packages," in *In Proceedings of the 2009 USENIX Annual Technical Conference (USENIX ATC*, 2009.

[18] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proceedings of the 14th ACM conference on Computer and communications security - CCS '07*, ACM Press, 2007.

[19] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Transactions on Software Engineering*, vol. 37, pp. 772–787, nov 2011.

[20] "Static code analysis." `https://www.owasp.org/index.php/Static_Code_Analysis`. Accessed: 2019-03-26.

[21] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering*, vol. 33, pp. 577–591, sep 2007.

[22] "Cyber grand challenge." `https://www.cybergrandchallenge.com/`. Accessed: 2019-03-26.

[23] N. L. Athos Ribeiro, Paulo Meirelles and F. Kon, "Ranking source code static analysis warnings for continuous monitoring of floss repositories." `https://www.oss2018.org/wp-content/uploads/2018/06/Athos-Ribeiro-oss.pdf`. Accessed: 2019-03-26.

[24] "National vulnerability database." `https://nvd.nist.gov/`. Accessed: 2019-03-26.

[25] "Open sourced vulnerability database." `http://www.osvdb.org`. Accessed: 2019-03-26.

[26] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates, "When a patch goes bad: Exploring the properties of vulnerability-contributing commits," in *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, IEEE, oct 2013.

[27] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, "VCCFinder," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*, ACM Press, 2015.

[28] "Common vulnerabilities and exposures." `https://cve.mitre.org/`. Accessed: 2019-03-26.

[29] "word2vec." `https://code.google.com/archive/p/word2vec/`. Accessed: 2019-03-26.

[30] A. Mani, "Solving text imputation using recurrent neural networks," 2015.

[31] S. Hochreiter, "Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München," 1991.

[32] J. F. Kolen and S. C. Kremer, *Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies.* IEEE, 2001.

[33] S. M. A. A. Mamun and J. Valimaki, "Anomaly detection and classification in cellular networks using automatic labeling technique for applying supervised learning," *Procedia Computer Science*, vol. 140, pp. 186–195, 2018.

[34] A. Ray, S. Rajeswar, and S. Chaudhury, "Text recognition using deep blstm networks," in *2015 Eighth International Conference on Advances in Pattern Recognition (ICAPR)*, pp. 1–6, Jan 2015.

[35] "Nist software assurance reference dataset." `https://samate.nist.gov/SARD/`. Accessed: 2019-03-26.

[36] "Bitcoin core." `https://github.com/bitcoin/bitcoin`. Accessed: 2019-03-26.