UDC 004.891.3
DOI: 10.15587/2706-5448.2021.233534
Article type «Reports on Research Projects»

Yevhenii Kubiuk,
Gennadiy Kyselov

COMPARATIVE ANALYSIS OF APPROACHES TO SOURCE CODE VULNERABILITY DETECTION BASED ON DEEP LEARNING METHODS

The object of research of this work is the methods of deep learning for source code vulnerability detection. One of the most problematic areas is the use of only one approach in the code analysis process: the approach based on the AST (abstract syntax tree) or the approach based on the program dependence graph (PDG).

In this paper, a comparative analysis of two approaches for source code vulnerability detection was conducted: approaches based on AST and approaches based on the PDG.

In this paper, various topologies of neural networks were analyzed. They are used in approaches based on the AST and PDG. As the result of the comparison, the advantages and disadvantages of each approach were determined, and the results were summarized in the corresponding comparison tables. As a result of the analysis, it was determined that the use of BLSTM (Bidirectional Long Short Term Memory) and BGRU (Bidirectional Gated Linear Unit) gives the best result in terms of problems of source code vulnerability detection. As the analysis showed, the most effective approach for source code vulnerability detection systems is a method that uses an intermediate representation of the code, which allows getting a language-independent tool.

Also, in this work, our own algorithm for the source code analysis system is proposed, which is able to perform the following operations: predict the source code vulnerability, classify the source code vulnerability, and generate a corresponding patch for the found vulnerability. A detailed analysis of the proposed system's unresolved issues is provided, which is planned to investigate in future researches. The proposed system could help speed up the software development process as well as reduce the number of software code vulnerabilities. Software developers, as well as specialists in the field of cybersecurity, can be stakeholders of the proposed system.

Keywords: *AST-based approaches, program dependence graph-based approaches, code analysis.*

Received date: 18.01.2021
Accepted date: 26.02.2021
Published date: 30.06.2021

How to cite
Kubiuk, Y., Kyselov, G. (2021). Comparative analysis of approaches to source code vulnerability detection based on deep learning methods. Technology Audit and Production Reserves, 3 (2 (59)), 19–23. doi: http://doi.org/10.15587/2706-5448.2021.233534

1. Introduction

Nowadays, information technologies are used in almost all spheres of human activity. As a result, the need for high-quality software is constantly growing. The larger the software product, the more critical vulnerability can be. Therefore, the task of source code analysis and searching for security vulnerabilities in it (e. g. buffer overflow, improper memory management) is very important for companies in the information technology industry.

The problem of analyzing program code is not a new one, and many techniques and tools have been created to solve it. The classical approach to source code vulnerability detection is an approach based on rules that are created by the expert [1, 2]. With the development of machine learning the algorithms were created, which use statistical and machine learning models to predict vulnerabilities in the code [3, 4]. The weakness of this approach is that it requires a technical expert, who would have to set up the system manually, for example: to create a dictionary of the language syntax, add information about grammatical structures, etc. These disadvantages are absent in models using the deep learning approach. An important advantage of deep learning is that the responsibility for identifying the necessary features for code analysis falls on the model itself.

Thus, *the object of research* in this paper is deep learning methods for source code vulnerability detection. *The aim of research is* to conduct a comparative analysis of existing deep learning in the tasks of source code vulnerability detection.

2. Methods of research

In this section, the application of deep learning methods for analyzing program code is reviewed. Source code analysis methods can be roughly divided into two groups: methods, based on the use of an AST (abstract syntax tree), and methods, based on the program's dependence graph.

ISSN 2664-9969

An AST describes the syntactic structure of program code. AST is often used in tasks of analyzing and correcting program code. AST nodes represent various syntax elements of the programming language, such as variables, functions, operators, and so on. Any valid program code has its own representation in the form of an AST, just as any AST can be converted to a valid program code. In practice, to get an AST from code, special tools are used. The tool can be either a part of the compiler (or interpreter), as it is done in Clang [5], or it can be a separate tool that can work with several programming languages and provide a universal AST, for example, Babelfish [6].

PDG (program dependence graph) describes library functions and API calls. It allows to detect and correct erroneous calls. The dependence graph can be divided into two categories: the data flow graph (DFG) and the control flow graph (CFG).

3. Research results and discussion

3.1. AST-based methods. In [7], AST is used for code clone detection using the supervised learning method. The algorithm for detecting duplicate code is as follows:

1) based on the source code, AST's are built;

2) ASTs are converted to binary trees;

3) using the word2vec model [8], each node of the binary tree is vectorized;

4) vectorized binary trees are fed to the input of the Siamese neural network [9], which uses LSTM [10] as a subnet.

In [7], various models with different hyperparameters were compared, and the usefulness of pre-trained embeddings for learning-based approaches in the context of software engineering was experimentally demonstrated. The disadvantage of the proposed algorithm is the use of a binary tree instead of the original AST. Representing an AST as a binary tree increases the height of the tree, and potentially leads to a loss of semantic connection between program code entities. This may primarily affect the location of node embeddings in the resulting hyperspace, as well as the accuracy of the neural network.

In [11], AST is used to predict defects in software using a supervised learning approach. In this paper, the authors proposed a tree structure of the LSTM network – Tree-LSTM. The system works as follows:

1) based on the source code, an AST is built;

2) each AST node is vectorized using ast2vec [11];

3) AST with vectorized nodes is fed to the input of Tree-LSTM, which generates a vector representation for the entire AST;

4) in the vector representation, the AST is fed to the input of traditional models for binary classification (Logistic Regression, Random Forests) in order to determine the vulnerability of the code.

The main advantage of the Tree-LSTM is that the model automatically determines the features during the process of learning. One of the drawbacks of the algorithm is that the AST is built for each project file separately, which means that the relationship between different files is not taken into account. According to the authors of [12], this can potentially lead to a lower accuracy of the system. Also, one of the drawbacks of the system is that the resulting binary tree representation of AST has a large height. The use of such trees in the training a neural network can lead to the problem of vanishing gradient [13].

The authors in [14] aimed to eliminate the vanishing gradient problem when training neural network models based on AST. For this purpose, the ASTNN model was developed, which generated its vector representation based on the AST. In contrast to the previous work, the authors split the AST into logically atomic blocks and translated them into a vector representation. Based on the set of vectors of one AST, a single vector representation of AST was generated. This model was tested for code classification tasks, as well as for code clone detection tasks. The model uses recurrent neural networks. The disadvantages of this approach include the potential loss of associative information between atomic AST blocks.

Table 1 presents a comparative analysis of code analysis solutions using AST-based approaches.

Table 1

Comparative analysis of source code analysis approaches using AST

Paper	Technology	Problem	Metrics	Language
[1]	Siamese LSTM	Code clone detection	AUC – 0.993	Java
[4]	Tree-LSTM, Logistic Regression	Source code vulnerabilities detection	F1 – 0.52 Precision – 0.36 Recall – 1.0 AUC – 0.59	C
	Tree-LSTM, Random Forest		F1 – 0.92 Precision – 0.93 Recall – 0.93 AUC – 0.99	
[5]	ASTNN	Code clone detection	Precision – 0.989 Recall – 0.927 F1 – 0.955	C
		Source code vulnerabilities detection	Accuracy – 0.982	

According to the data in Table 1, it is possible to conclude that using AST models to solve the problem of source code vulnerabilities detection demonstrates a higher accuracy, comparing to systems that use AST-based models only for code vectorization tasks.

3.2. Methods, based on program dependence graph. In [15], the authors demonstrate a method for representing source code in the form of a data flow graph, while preserving the semantic structure of the program. With the approach of using a data flow graph, authors solved the problems of incorrect naming of variables in the code (VarNaming) [16], as well as the problem of incorrect use of variables (VarMisuse) [15]. GGNN [17], a model specially adapted for working with graphs, was used as a neural network model. The disadvantage of this representation of the source code is that the resulting graph is too large. A large number of nodes and links between them affected the accuracy of the model – 52.6 % for VarNaming and 85.5 % for VarMisuse.

In [18] authors perform source code vulnerability detection using the adjacency matrix of the control flow graph. As an atomic unit for the graph, small code blocks of the same type were selected (in terms of syntax), which, in turn, were translated into a vector representation using the word2vec model. A subtype of convolutional neural network (CNN) [19] was used. TextCNN [20] was used as a neural network. The disadvantages of this work can be

attributed to the fact that the author did not compare the performance of the model using information about data flow and without. The use of a data flow graph would provide additional information about the associative relationship of program code entities, which could potentially improve the prediction results.

In [21], the problem of source code vulnerability detection (CWE-119, CWE-399 [22]) was solved using the data flow graph. The authors formed the so-called code gadget based on the list of potentially vulnerable library functions and the data flow graph. The code gadget consisted of lines of code that refer to the arguments or return value of a potentially vulnerable target function (depending on the function type). For functions that are used for memory allocation (malloc/new), the code gadget was built by tracking the return values in the data flow graph. For functions that are designed to deallocate memory (free/delete), the code gadget was built for their arguments. The disadvantage of such a system is the lack of information from the control flow, as noted by the authors of [23].

Also, the accuracy of the model is significantly affected by the size of the code gadget. In real projects, a variable that holds allocated memory and was created in one function can be passed to many other functions as an argument. The authors of the study [24] conducted a comparative analysis of the performance of models with different sizes of code gadgets. In their solution, an empirically selected nesting limit was used to traverse the data flow tree, which resulted in an increase in prediction accuracy compared to [21].

The authors of [23] extended the functionality of [21] and [24] by adding information from the control flow graph. Also, in contrast to [21] and [24], which focused on detecting buffer overflow-related vulnerabilities (CWE-119) and resource management-related vulnerabilities (CWE-399), the authors of [23] detected 126 different types of vulnerabilities. According to the study [23], the use of BGRU as a model for predicting source code vulnerabilities proved to be better than the use of BLSTM [25]. The main limitation of the algorithms in works [21–23] is that they are based on a previously defined list of potentially vulnerable library functions and APIs. This means that the model is not able to detect a vulnerability that is not related to the library function of a certain API call. Also, the question of optimal code gadget size was raised by the authors in [24]. Although the problem of the size of the code gadget was raised by the authors of [24], the selection of the optimal nesting size for generating the code gadget remains an important problem.

In [26], the authors created a system for source code vulnerability detection based on data flow and control flow graphs. First, the so-called syntax vulnerability candidate (SyVC), was built based on the AST. Then the semantic vulnerability candidate (SeVC) was generated on its basis. In this paper, the authors demonstrated the effectiveness of this approach, and also presented a comparison of the use of various neural network models in terms of source code vulnerability detection task. Later, in this work [27], the authors showed that the use of SySeVC for tasks of source code vulnerability detection is effective not only for source code in its original appearance, but also for intermediate representation (IR) of the source code. Intermediate representation can be obtained by using LLVM tools [28].

Table 2 provides a comparative analysis of code analysis solutions using an approach based on the program dependence graph.

Table 2

Comparative analysis of source code analysis approaches using PDG

Paper	Technology	Problem	Metrics	Language
[8]	GGNN	VarNaming	Accuracy – 0.536 F1 – 0.658	C#
		VarMisuse	Accuracy – 0.855 AUC – 0.980	
[11]	TextCNN	Source code vulnerabilities detection	AUC – 0.82	C/C++
[21]	BLSTM	Source code vulnerabilities detection	Accuracy – 0.908 Precision – 0.92 F1 – 0.934	C/C++
[23]	BLSTM	Source code vulnerabilities detection	Accuracy – 0.967 Precision – 0.909 F1 – 0.924	C/C++
	BGRU		Accuracy – 0.968 Precision – 0.919 F1 – 0.925	
[24]	BLSTM	Source code vulnerabilities detection	Accuracy – 0.92	C/C++
[19]	BGRU	Source code vulnerabilities detection	Accuracy – 0.960 Precision – 0.88 F1 – 0.844	C/C++
[20]	BLSTM	Source code vulnerabilities detection	Accuracy – 0.977 Precision – 0.985 F1 – 0.952	C/C++ (LLVM IR)
	BGRU		Accuracy – 0.988 Precision – 0.982 F1 – 0.972	

According to the presented data, it is possible to conclude that using BGRU for the task of source code vulnerability detection gives a better result than using alternative neural network models. This statement is true for both problems of finding vulnerabilities in the original source code and for problems of finding vulnerabilities in the intermediate representation of the source code (e. g. LLVM).

3.3. Source code analysis system. As a result of the analysis, an algorithm for source code vulnerability detection is proposed (Fig. 1).

There are several ways to generate an AST: first, build an AST based on an intermediate source code representation using LLVM, or second way – build an AST using universal AST construction tools. Then, using the program dependence graph, the code gadget is constructed, which after being translated into a vector representation is fed into the neural network model.

The next step of the algorithm is to classify whether the code gadget is vulnerable or not. In case if code gadget is vulnerable, the algorithm should classify the type of vulnerability. The final step of the algorithm is to generate a recommendation of a possible fix of the vulnerability. There are still several open questions in this study:

1. Selection of the AST construction method.
2. Size of the generated code gadget.
3. Model for classifying vulnerabilities.
4. Model for generating patches.

ISSN 2664-9969

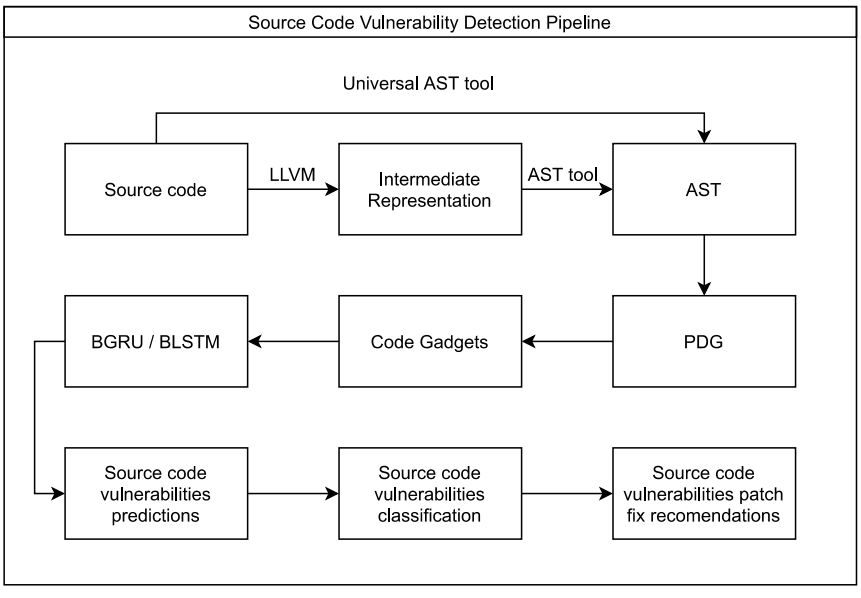

Fig. 1. Algorithm for source code vulnerability detection

4. Conclusions

In this paper, deep learning methods for source code vulnerability detection ARE ANALYZED. Approaches to analyzing the source code were divided into two categories: AST-based methods and PDG-based methods. When using approaches, based on AST, the authors focus on the tasks of converting the original AST into a structure that would be able to highlight the syntax features of the source code. Also, in theirs works, the authors pay attention to the selection of a deep learning model that would be able to solve the task correctly without losing the associative connection of program code entities. The dependence graph approach allows operating with information obtained from the control flow and data flow. This approach provides the neural network model with additional information about the semantic relationship of source code entities. However, this approach has its own drawbacks – often the code sections semantically associated with a variable or function call are quite large, which leads to lower prediction accuracy. The problem of the universality of this approach relatively to different programming languages remains important. At the current stages of research, the differences in prediction accuracy between the original code and the intermediate LLVM representation are insignificant. Also, in this work, an algorithm for analyzing source code was proposed. A tool that would work according to the presented algorithm would help to speed up the software development process, as well as reduce the number of source code vulnerabilities. Software developers, as well as specialists in the field of cybersecurity, can be stakeholders of the proposed system.

References

1. Prähofer, H., Angerer, F., Ramler, R., Lacheiner, H., Grillenberger, F. (2012). Opportunities and challenges of static code analysis of IEC 61131-3 programs. *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*. IEEE, 1–8. doi: http://doi.org/10.1109/etfa.2012.6489535
2. Lee, M., Cho, S., Jang, C., Park, H., Choi, E. (2006). A rule-based security auditing tool for software vulnerability detection. *2006 International Conference on Hybrid Information Technology*. IEEE, 2, 505–512. doi: http://doi.org/10.1109/ichit.2006.253653
3. Turhan, B., Kocak, G., Bener, A. (2009). Data mining source code for locating software bugs: A case study in telecommunication industry. *Expert Systems with Applications, 36 (6)*, 9986–9990. doi: http://doi.org/10.1016/j.eswa.2008.12.028
4. Murakami, H., Hotta, K., Higo, Y., Igaki, H., Kusumoto, S. (2013). Gapped code clone detection with lightweight source code analysis. *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 93–102. doi: http://doi.org/10.1109/icpc.2013.6613837
5. *Clang: A C Language Family Frontend for LLVM*. Available at: https://clang.llvm.org/
6. Babelfish. *GitHub*. Available at: https://github.com/bblfsh
7. Büch, L., Andrzejak, A. (2019). Learning-based recursive aggregation of abstract syntax trees for code clone detection. *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 95–104. doi: http://doi.org/10.1109/saner.2019.8668039
8. Mikolov, T., Chen, K., Corrado, G., Dean, J. (2013). *Efficient estimation of word representations in vector space*. Available at: https://arxiv.org/abs/1301.3781
9. Bromley, J., Guyon, I., LeCun, Y., Säckinger, E., Shah, R. (1993). Signature verification using a «Siamese» time delay neural network. *Advances in neural information processing systems, 6*, 737–744.
10. Hochreiter, S., Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation, 9 (8)*, 1735–1780. doi: http://doi.org/10.1162/neco.1997.9.8.1735
11. Dam, H. K., Pham, T., Ng, S. W., Tran, T., Grundy, J., Ghose, A. et. al. (2018). *A deep tree-based model for software defect prediction*. Available at: https://arxiv.org/abs/1802.00921
12. Guan, Z., Wang, X., Xin, W., Wang, J., Zhang, L. (2020). A survey on deep learning-based source code defect analysis. *2020 5th International Conference on Computer and Communication Systems (ICCCS)*. IEEE, 167–171. doi: http://doi.org/10.1109/icccs49078.2020.9118556
13. Hochreiter, S. (1998). The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems, 6 (2)*, 107–116. doi: http://doi.org/10.1142/s0218488598000094
14. Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X. (2019). A novel neural source code representation based on abstract syntax tree. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 783–794. doi: http://doi.org/10.1109/icse.2019.00086
15. Allamanis, M., Brockschmidt, M., Khademi, M. (2017). *Learning to represent programs with graphs*. Available at: https://arxiv.org/abs/1711.00740

16. Allamanis, M., Barr, E. T., Bird, C., Sutton, C. (2014). Learning natural coding conventions. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 281–293. doi: http://doi.org/10.1145/2635868.2635883
17. Li, Y., Tarlow, D., Brockschmidt, M., Zemel, R. (2015). *Gated graph sequence neural networks*. Available at: https://arxiv.org/abs/1511.05493
18. Harer, J. A., Kim, L. Y., Russell, R. L., Ozdemir, O., Kosta, L. R., Rangamani, A. et. al. (2018). *Automated software vulnerability detection with machine learning*. Available at: https://arxiv.org/abs/1803.04497
19. LeCun, Y., Haffner, P., Bottou, L., Bengio, Y. (1999). Object recognition with gradient-based learning. *Shape, contour and grouping in computer vision*. Berlin, Heidelberg: Springer, 319–345.
20. Kim, Y. (2014). *Convolutional neural networks for sentence classification*. Available at: https://arxiv.org/abs/1408.5882
21. Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S. et. al. (2018). VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. *Proceedings 2018 Network and Distributed System Security Symposium*. doi: http://doi.org/10.14722/ndss.2018.23158
22. *CWE – Common Weakness Enumeration*. CWE. Available at: https://cwe.mitre.org/
23. Li, Z., Zou, D., Tang, J., Zhang, Z., Sun, M., Jin, H. (2019). A Comparative Study of Deep Learning-Based Vulnerability Detection System. *IEEE Access, 7*, 103184–103197. doi: http://doi.org/10.1109/access.2019.2930578
24. Chrenousov, A., Savchenko, A., Osadchyi, S., Kubiuk, Y., Kostenko, Y., Likhomanov, D. (2019). Deep learning based automatic software defects detection framework. *Theoretical and Applied Cybersecurity, 1 (1)*. doi: http://doi.org/10.20535/tacs.2664-29132019.1.169086
25. Schuster, M., Paliwal, K. K. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing, 45 (11)*, 2673–2681. doi: http://doi.org/10.1109/78.650093
26. Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z. (2021). SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 1–1. doi: http://doi.org/10.1109/tdsc.2021.3051525
27. Li, Z., Zou, D., Xu, S., Chen, Z., Zhu, Y., Jin, H. (2021). VulDeeLocator: A Deep Learning-based Fine-grained Vulnerability Detector. *IEEE Transactions on Dependable and Secure Computing*, 1–1. doi: http://doi.org/10.1109/tdsc.2021.3076142
28. *The LLVM Compiler Infrastructure Project*. Available at: https://llvm.org/

Yevhenii Kubiuk, *Department of System Design, National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute», Kyiv, Ukraine, e-mail: eugen.kubiuk@gmail.com, ORCID: http://orcid.org/0000-0002-7086-0976*

Gennadiy Kyselov, *PhD, Department of System Design, National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute», Kyiv, Ukraine, e-mail: g.kyselov@gmail.com, ORCID: https://orcid.org/0000-0003-2682-3593*