An Enhanced Deep Learning methods for Defect Analysis in source code

Shalinibharide, NAVYA ANGATI,NISHCHALA THAKUR

Assoc.prof
Department: CSE
Visakha Institute of Engineering & Technology,
Division,GVMC,Narava, Visakhapatham, Andhra Pradesh.

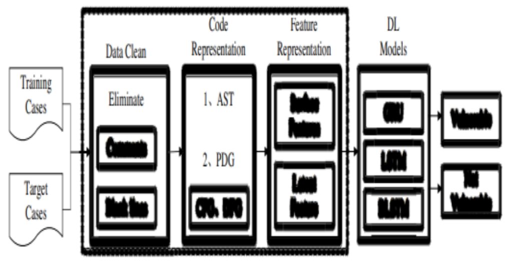
Abstract— ***A slew of software apps have flooded our everyday routines as information technology has advanced rapidly. It's certain that a lot of code will be generated during the development of these applications. Researchers in the academic sector are interested in learning how to identify and evaluate numerous faults in the source code, such as API/Function call mistakes, array abuse, and expression syntax error, among others. Researchers have attempted to employ deep learning algorithms in AI to automatically extract and evaluate aspects of source code because artificial intelligence (AI) technology has achieved remarkable achievements in the areas of image processing and natural language processing. As a result, we take a look at the most current deep learning-based algorithms for analysing source code defects. The automated extraction of source code fault characteristics is possible using deep learning-based code defect analysis approaches, as opposed to conventional methods. As a result, human specialists are no longer required to pre-define code features, which helps to reduce mistakes caused by people. An intriguing and hard development path is the use of AI in defect analysis of source code. We feel this has a wide range of potential.***

Keywords- vulnerability detection; deep learning; AST; PDG; source code defect analysis

INTRODUCTION

People's everyday routines have changed dramatically as a result of the fast advancement of information technology. Although a plethora of apps have made our lives easier, bugs in software often pose unknown security dangers. Moreover, with the rise of open-source initiatives, code reuse is no longer an unusual practise. API/Function call mistakes, array abuse, and expression syntax issues, among other source code flaws, are critical to preventing destructive assaults by hackers and ensuring user data security. Many academics and specialists are working to enhance the current methods for detecting and analysing source code defects, both in academia and in industry. Natural language processing (NLP) approaches may be used to analyse source code defects since the language of source code is fundamentally a textual language. A number of methods have been used by academics to extract certain syntactic aspects or code patterns from the source code, such as data dependency and function dependency. In the field of source code defect analysis (SCDA), researchers apply both traditional machine learning techniques and deep learning algorithms. SCDA models have been constructed using Machine Learning [1]. The majority of ML-based SCDA approaches rely on the extraction and comparison of features from both faulty and non-defective code. ML-based approaches collect important characteristics from the source code and use one or more classifiers to assess the recovered features to determine if the source code includes vulnerabilities. Naive Bayes, SVM, and Random Forest are just a few of the common machine learning classifiers. The standard ML-based SCDA approaches, on the other hand, have certain glaring flaws. Notably, human specialists are required to predefine aspects of source code, such as vocabulary and grammatical structure information, in order for this sort of approach to work. Because of this, DL technology has been brought into the area of SCDA. When compared to more conventional machine learning techniques. Automated extraction of source code features using DL algorithms may be accomplished via the use of several convolutional and activation layer layers. When it comes to machine learning, there are no human specialists involved, which essentially eliminates the risk of human mistake [2]. Essentially, source code is a kind of textual data. Regardless of whether it is a machine learning algorithm or a deep learning method, the ability to represent the source code is a necessity for implementing SCDA. Previous approaches used token fragmentation and information retrieval to accomplish SCDA, such as clone detection [3]-[6], vulnerability prediction [7]-[8], bug location [9], and so on. But this code representation approach has the problem of being unable to take into account the intricate structural information included in the source code. Some DL-based approaches use alternative representations of source code before further analysis, such as Abstract Syntax Trees (AST), Bytecodes, Program Dependency Diagrams (PDD), etc. For the purpose of this work, we focus on current DL-based approaches for source code defect analysis (SCDA), which are particularly useful for automated code defect analysis and vulnerability identification.

In general, there are three types of DL-based SCDA techniques to choose from: AST-based, PDG-based, and other DL-based. When compared to other techniques of representation, the AST



In Figure 1, you can see an overview of the process of detecting source code defects using deep learning.

In Section II, we provide an overview of the fundamental principles and models of deep learning. AST-based, PDG-based, and other DL-based approaches are discussed in length in Section III of this paper. Section IV sums up the most widely utiliseddatasets and assessment measures. Section V concludes with a look to the future and the conclusion.

II. CONCEPTS OF DEEP LEARNIN

In the field of machine learning algorithms, deep learning is a relatively recent development that uses artificial neural networks to solve issues. To begin, the DL approach was used to classify images, but it has since been extended to a wide range of additional applications in the domains of computer vision, natural language processing (such as voice recognition and conversation robots), and pattern identification in computer vision (NLP).

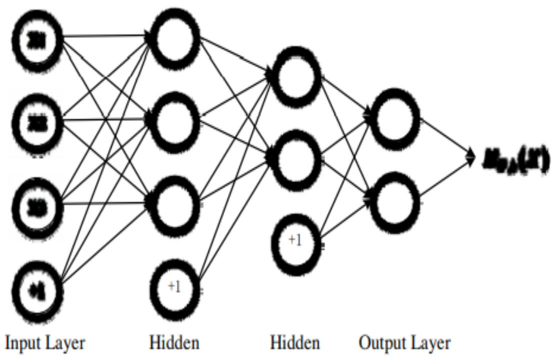


Figure 2. Diagram of artificial neural networks.

There are many layers in an ANN [10], including the input layer, the hidden layer, and the output layer, as seen in Figure 2. In addition, training and testing are also components of the issue solution process based on deep learning. Instead of requiring experts to pre-define the features of defect source code as in traditional SCDA methods, feature extraction through deep leaning can reduce human labour and effectively avoid human error in the manual definition of features.

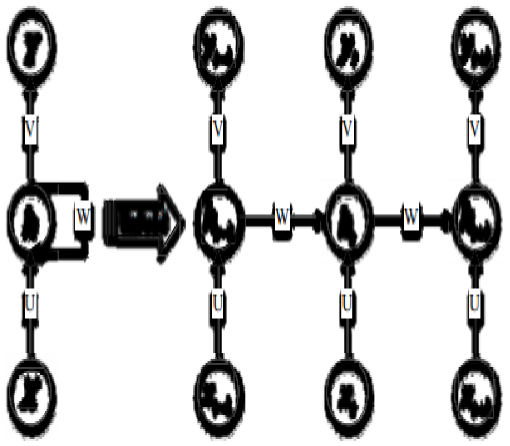


Figure 3. Diagram of Recurrent Neural Networks.

Since most of the neural network models in the area of natural language processing are equally relevant to SCDA since the source code is essentially textual information. Recurrent Neural Networks (RNN) have a simple design as seen in Figure 3 [11]. The vectors of tokens of source code may be used as input to RNNs, and the encoded feature vector corresponding to the input code fragment is typically the final output. LSTM [12] and GRU [13] are two more great deep learning models that may be used for SCDA.

DEEP LEARNING-BASED SCDA METHOD

In this part, we'll take a look at some of the more recent SCDA DL-based projects. To begin, we'll look at both AST and PDG approaches separately in two separate subsections. In the next section, we provide several more DL-based strategies for identifying vulnerabilities.

AST-Based Methods

Analysis of source code defects is often performed using an Abstract Syntactic Tree (AST), a tree representation of the source code's abstract syntax structure. The nodes in the AST tree generally reflect a structure in the source code when used to describe the source code. In particular, the parenthesis in nested statements, which are not shown as nodes, will not be represented by the AST. There are several alternative aggregation approaches for code clone detection that have been studied in [14] by the authors. For code clone detection, in [14], an AST-based recursive neural network is suggested using static source code. An exploratory paper like this one [14] may be considered since it not only studies model selection and hyperparameters, but also studies the effect of pretrained embeddings representing nodes in ASTs. To solve the issue of class imbalance, we've found that error scaling works well. In the preprocessing step, the original AST of source code is transformed to a binary tree, which may increase AST depth and lose essential association information between nodes. This is an evident disadvantage of this technique. Tree-LSTM (Tree-LSTM) was suggested by authors in [15] as a way to forecast bugs in source code. For the Tree-LSTM model in [15], the AST representation of the source file was used. Method [15] constructs an AST from the root of the AST since it represents an entire source file. The benefit of using the Tree-LSTM approach is that it can automatically learn all of the characteristics. However, the Tree-LSTM model's limitations are also readily apparent. A potential consequence of [15]'s AST representation of source code is that connected information between code fragments may be lost. For source code representation, an AST-based Neural Network (ASTNN) was developed. Authors in [16] attempted to break each big AST into a series of little statements, and employed a bidirectional RNN model to integrate the chain of statement vectors into a larger AST. Most importantly, it sought to break apart the huge ASN into smaller statements and then encoded vectors in terms of statement trees, which is the key contribution to the ASTNN technique It is possible that the relationship between statement trees may have been lost by integrating the encoded vectors of the succession of statement trees into one vector representation, however this was not the case.

PDG-Based Methods

Researchers strive to build PDGs and extract information from them because of the fact that major source code faults are common during the process of function call. Two types of PDG are routinely used: data flow graphs (DFGs) and control flow diagrams (CFDs) (CFGs). It was recommended that graphs be used to describe both the semantic and syntactic structure of source code since the existing approaches did not consider utilising the code's known syntax as a priori knowledge, such as the same variable or function at remote places. With regard to "VARNAMING" and "VARMISUSE" tasks [17,18], in order to encode the programme graph, the graph-based deep learning approaches, built upon Graph Neural Networks (GGNN), were used. When it comes to constructing programme graphs of source code, the technique in [17] provides a detailed explanation of the process, but it also shows how to extend deep learning models training to enormous graphs. In addition, the route constraint issue is a study area that demands particular attention when employing Fuzzing approach for software testing. Since most Fuzzers can't handle route constraints including deeply nested conditional statements effectively, writers in [19] developed a Matryoshka technique to deal with this issue. In the Matryoshka approach, all of the control flow dependent conditional statements are found using post-dominator trees, both inside and between procedures. Taint flow dependence was removed from control flow dependent conditional statements in order to simplify the number of calculations and prevent modifying every byte. With the Matroshka approach, numerous route constraint-solving methodologies may be utilised, and the gradient descent algorithm is used to identify solutions. The Matryoshka approach obviously considers both the control and data dependence of the source code in SCDA, even if the authors in [19] did not mention it. Prior to data collection, a static analyzer was used to identify open-source routines and build a big data set. And the C/C++ source code's Control Flow Graphs (CFGs) were utilised to extract features. Differently, each node in the function's CFG represents a fundamental block rather than a statement in a function. Method in [20] may be characterised in two ways: first, build-based techniques and source-based methods were utilised for feature extraction; second, both random trees classifier of standard ML algorithm and TextCNN-based model of DL algorithm were examined for source code evaluation. The Data Flow Graphs (DFGs) of source code, however, are not examined in this manner. In addition, researchers in [21] employed a deep learning-based detection system (VulDeePecker) for vulnerability identification since the attributes of vulnerabilities were described by human experts in an existing technique. VulDeePecker uses Data Dependency as the semantic information in source code, as opposed to method in [20]. VulDeePecker's key disadvantage is that only the DFG in the PDG is addressed, and the function of the CFG is not studied, which raises the question of whether the semantic information of CDG may enhance the performance of vulnerability detection, as the authors point out. The CFG and DFG were utilised to extract source code information from SySeVR [23]. Deep neural networks (Bidirectional-LSTM) were used to investigate how to encode vulnerability patterns and accomplish SCDA as a result of the successful use of DL in image processing [23]. First, the AST was used to extract syntactic information from source code, and then, the CFG and DFG of source code were used to extract semantic information from source code. BLSTM is exclusively trained on semantic information taken from PDG in SySeVR's BLSTM, which may raise the issue of whether syntactic and semantic information retrieved from PDG are complimentary or not. Authors in [22] performed a comparison research to analyse the effect of numerous aspects, such as the selection of DL models and the affect of different imbalanced data processing approaches, on the vulnerability identification process. According to [22], the data persistence and control reliance of PDG objects are two separate ways that semantic information is manifested in a programme. Similar to DL models, this technique is limited by the characteristics it has learnt.

Other DL-Based Methods

Multiple approaches for classifying vulnerabilities were used by writers in [24], including TF-IDF, IG and Deep Neural Networks, to lower the risk of attack and better manage the vulnerabilities (DNN). For each word in a vulnerability description, the TF-IDF is used to calculate its frequency and weight; the IG is used to identify features; and the DNN model is utilised as a vulnerability classifier. One of the advantages of the approach in [24] is that it compared classification performance between DNN and SVM, Naive Bayes and KNN, confirming the usefulness of the deep learning model under certain tasks. To name only a few of the topics covered by the SCDA, there is code cloning and vulnerability categorization and mining. With the concept of "transfer learning," researchers may apply various DL-based approaches to certain SCDA paths and see whether they provide any surprising outcomes.

EVALUATION METHODS

DataSets National Vulnerability Database (NVD) [25] and Software Assurance Reference Dataset (SCRD) [26] are the most often utilised vulnerability data sources. Vulnerability management principles developed by the U.S. government in 2000 are housed in the National Vulnerability Database (NVD). The Common Vulnerabilities and Exposures (CVE) List was used to build this dataset. CVEs that have been published to the CVE Dictionary are typically the ones that are analysed by this tool. As a result, NVD will be the first to get any CVE updates. Users and researchers may benefit from this dataset, which includes a list of known security issues. The SARD dataset covers a wide variety of test cases culled from a variety of various places, including industrial programmes, synthetic data, and academic research. The applications in SARD may be split into three categories: "good" programmes, "bad" programmes, and "mixed" programmes, which include both the vulnerability and the patched version. 3. Measuring First, we'll review some of the most often used assessment phrases and meanings. SCDA assessment measures are then introduced.... Samples that are susceptible in fact and are recognised as such are referred to as "true positives." A "true negative" (TN) is a sample that has been shown to be free of vulnerability. Non-vulnerable samples that are incorrectly identified as susceptible. False Negative (FN): Samples that are truly susceptible but are incorrectly identified as invulnerable. Percentage of projected samples that were properly predicted (ACC) The following is the text from the transcription: TP TN Prediction precision (P) is the ratio of properly predicted positive samples (TP) to the total number of anticipated positive samples. P TP FP TP Remember (R): The proportion of properly anticipated positive samples to the total susceptible samples. R TP TP FN FPR: The percentage of false positive samples (FP) compared to the total number of samples that are truly susceptible. FPR FP is an abbreviation for "Full Total samples that are susceptible to false negatives, as defined by the False Negative Rate, or FNR. FNR FN FN TP

FUNDING STATEMENT

The National Natural Science Foundation of China (NSFC) under Grant [U1736110], partially supported by the NSFC under Grant [U1836209], partially supported by the NSFC under Grant [U1636115], partially supported by the National Key Research and Development Program of China (2018YFB0804101) under Grant [2018YFB0804101]

ACKNOWLEDGMENT

Anonymous reviewers have provided the writers with valuable feedback on the manuscript's modification.

CONCLUSION AND FUTURE WORK

Deep learning-based (DL-based) defect analysis of source code is reviewed in this study (SCDA). For the most part, DL-based SCDA approaches fall into one of three basic groups: Program Dependency Graph (PDG)-based (PDG-based) approaches, as well as other DL-based methods. Deep neural networks are used in the AST-based technique. As far as source code analysis is concerned, the PDGs have been included since function call processes are where most severe code faults occur. There are two popular ways to encode semantic information in source code: the DFG and CFG. While classic ML-based SCDA approaches need professionals to describe aspects of software vulnerabilities, the DL-based SCDA methods do not, and the analysis or detection of source code faults is frequently automated. It is clear from the preceding discussion that DL technologies and approaches in the area of source code defect analysis have significant development opportunities. However, there are still certain issues that can't be overlooked in this sort of approach.. Code defect analysis approaches are more successful if they are based on accurate data. This is why it is necessary to gather and pre-process the code defect data in the future. AI-based source code defect analysis is a fascinating and demanding research area that should be further investigated.

REFERENCES

[1] W. Xiaomeng, Z. Tao, X. Wei, and H. Changyu, “A survey on source code review using machine learning,” in 2018 3rd International Conference on Information Systems Engineering (ICISE), May 2018, pp. 56–60.

[2] W. Xiao-meng, Z. Tao, W. Runpu, X. Wei, and H. Changyu, “Cpgva: Code property graph based vulnerability analysis by deep learning,” 2018 10th International Conference on Advanced Infocomm Technology (ICAIT), pp. 184–188, 2018.

[3] T. Kamiya, S. Kusumoto, and K. Inoue, “Ccfinder: a multilinguistic token-based code clone detection system for large scale source code,” IEEE Transactions on Software Engineering, vol. 28, no. 7, pp. 654– 670, July 2002.

[4] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, “Deep learning code fragments for code clone detection,” in 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), Sep. 2016, pp. 87–98.

[5] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, “Sourcerercc: Scaling code clone detection to big-code,” in Proceedings of the 38th International Conference on Software Engineering, ser. ICSE ’16. New York, NY, USA: ACM, 2016, pp. 1157–1168.

[6] H. Wei and M. Li, “Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code,” in Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17, 2017, pp. 3034–3040. [7] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto,“An empirical comparison of model validation techniques for defect prediction models,” IEEE Transactions on Software Engineering, vol. 43, no. 1, pp. 1–18, Jan 2017.

[8] M. D’Ambros, M. Lanza, and R. Robbes, “Evaluating defect prediction approaches: A benchmark and an extensive comparison,” Empirical Softw. Engg., vol. 17, no. 4-5, pp. 531–577, Aug. 2012.

[9] J. Zhou, H. Zhang, and D. Lo, “Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports,” in 2012 34th International Conference on Software Engineering (ICSE), June 2012, pp. 14–24.

[10] G. E. Hinton, S. Osindero, and Y.-W. Teh, “A fast learning algorithm for deep belief nets,” Neural Comput., vol. 18, no. 7, pp. 1527–1554, Jul. 2006.

[11] T. Mikolov, M. Karafiát, L. Burget, J.澂 ernock`y, and S. Khudanpur, “Recurrent neural network based language model,” in Eleventh Annual Conference of the International Speech Communication Association, p. 1045.

[12] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” Neural Comput., vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [13] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using rnn encoder–decoder for statistical machine translation,” in Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP). Association for Computational Linguistics, 2014, pp. 1724–1734.

[14] L. Bch and A. Andrzejak, “Learning-based recursive aggregation of abstract syntax trees for code clone detection,” in 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Feb 2019, pp. 95–104.

[15] H. K. Dam, T. Pham, S. W. Ng, T. Tran, J. Grundy, A. Ghose, T. Kim, and C. Kim, “A deep tree-based model for software defect prediction,” CoRR, vol. abs/1802.00921, 2018.

[16] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, “A novel neural source code representation based on abstract syntax tree,” in Proceedings of the 41st International Conference on

Software Engineering, ser. ICSE '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 783–794.

[17] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," CoRR, vol. abs/1711.00740, 2017. [18] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel, "Gated graph sequence neural networks," CoRR, vol. abs/1511.05493, 2016. [19] P. Chen, J. Liu, and H. Chen, "Matryoshka: fuzzing deeply nested branches," CoRR, vol. abs/1905.12228, 2019.

[20] J. A. Harer, L. Y. Kim, R. L. Russell, O. Ozdemir, L. R. Kosta, A. Rangamani, L. H. Hamilton, G. I. Centeno, J. R. Key, P. M. Ellingwood, M. W. McConley, J. M. Opper, S. P. Chin, and T. Lazovich, "Automated software vulnerability detection with machine learning," CoRR, vol. abs/1803.04497, 2018.