Check for updates

A Survey of the Software Vulnerability Discovery Using Machine Learning Techniques

Jian Jiang[1(✉)], Xiangzhan Yu[1,2], Yan Sun[2], and Haohua Zeng[2]

[1] School of Computer Science and Technology, Harbin Institute of Technology, Harbin, China
1356117066@qq.com, yxz@hit.edu.cn

[2] Institute of Electronic and Information Engineering of UESTC in Guangdong, Dongguan, China
289917845@qq.com, zhh0769@foxmail.com

Abstract. Nowadays, the study of vulnerability discovery has been attracted the widespread attention and the experts have proposed many different approaches in the past decades. To optimize the efficiency of the method, machine learning techniques are introduced into this area. In this paper, we provide an extensive review of the work in the field of software vulnerability discovery that utilize machine learning techniques. For the three key technologies of static analysis, symbolic execution and fuzzing in vulnerability discovery field, we first explain the basic principles respectively. Afterward, we review the research situation of software vulnerability discovery using machine learning techniques. Finally, we discuss both advantages and limitations of the approaches reviewed in the paper, and point out challenges and some uncharted territories in the three categories. In this paper, a brief study of the software vulnerability discovery using machine learning techniques is given, which is helpful to carry out the follow-up research work.

Keywords: Vulnerability discovery · Machine learning · Static analysis · Deep learning · Symbolic execution · Fuzzing

1 Introduction

In the current cyberspace, the number of serious vulnerabilities discovered in software is on the rise. The severity of these vulnerabilities has varying degrees, depending on factors such as exploitation complexity and attack-surface [1]. Numerous vulnerabilities seriously undermine the security of computer systems and IT infrastructure of companies and individuals. For instance, a vulnerability in the server message block (SMB) protocol exploited by the WannaCry ransomware have affected a wide range of systems and millions of users worldwide [2].

There are many software vulnerability discovery techniques, and the classification methods are also various. It can be divided into manual, automatic and semi-automatic depending on the degree of automation. From the point of view of code execution, it can be divided into dynamic analysis, static analysis and hybrid analysis. It also can be divided into black box test, white box test and gray box test depending on whether the software is open source. There are crossovers between the various categories [12]. Now, the main vulnerability discovery techniques that the paper mentioned is static analysis, symbolic execution and fuzzing. Symbolic execution both belong to the software dynamic analysis under the operating angle and can also be White box test divided by code openness, fuzzing is both dynamic analysis and black box test.

No matter what kind of the vulnerability discovery techniques, machine learning provides a new opportunity for intelligent, effective, and efficient approach. In this paper, we present an extensive review about static analysis, symbolic execution and fuzzing that utilize machine learning techniques. First, we explain the basic principles of static analysis. Afterward, we particularly review the research using machine learning techniques. In the end, we discuss their advantages and limitations of the approaches the reviewed papers proposed, point out the challenges in the field and some uncharted domains to inspire future work in this emerging research area.

2 Static Analysis Using Machine Learning Techniques

2.1 Principles of Static Analysis

Static analysis is the process of evaluating a system or component based on its form, structure, content, or documentation, which does not require program execution [3]. Based on targets' type, static analysis can be classified as source code analysis and binary analysis. In practice, their methods are the same, but the latter is more difficult [12].

Static analysis techniques include rule matching, data-flow analysis, control-flow analysis, program dependence analysis, information-flow analysis, static slice, abstract interpretation, model checking and theorem provers [3]. Nowadays, many research work on the static analysis using machine learning techniques have been done, and the result is inspiring.

2.2 Summary of Recent Works

Scandariato et al. [4] treated a programming language as a native language and analyzed the source code by means of text mining techniques. The proposed method is mainly based on the Bag-of-Words technique, where software component is seen as a series of terms with associated frequencies. The paper analyzed 20 "apps" include 182 releases for the Android OS platform using five well known machine learning algorithm: Decision Trees, k-Nearest Neighbor, Naive Bayes, Random Forest and Support Vector Machine (SVM). The best results are obtained by Naive Bayes and Random Forest. The paper used a commercial program vulnerability analysis solution to provide the labels of training dataset and performed three experiments. In the first experiment, the authors built both Naive Bayes model and Random Forest model based on the first version of each application. The other two experiments built a prediction model with subsequent-version and cross-project applications. According to the reported results, the first and second experiment's prediction is acceptable, yet the prediction of the last experiment is not.

Shin et al. [5] first apply artificial neural networks to binary analysis and tackle the problem of function identification. The central challenge is the lack of high-level semantic structure within binaries, as compilers discard it from the source code. Therefore, to recognize functions is the most important step in binary analysis. The paper use "one-hot encoding" to convert a byte into a vector, push it as the input of the neural network and used bidirectional models with RNN hidden units. Besides that, the paper also optimize the result with stochastic gradient descent and "rmsprop" method. The paper shows that recurrent neural networks can solve recognizing functions more efficiently than the previous state-of-the-art method in binaries analysis.

Perl et al. [6] presented a new method to identify vulnerability contributing commits. The authors combine code-metric analysis with metadata contained in code repositories. To evaluate the effectiveness, the authors first conduct a dataset containing 66 C/C++ GitHub projects with 170860 commits including 640 vulnerability contributing commits mapped to relevant CVE IDs. The paper selected project, author, commit, file as GitHub rich meta-data, and extracted the code-churn and developer-activity as the code-metrics. Afterward, the paper created a generalized Bag-of-Words model using the above features and trained a classifier of vulnerability contributing commits using linear SVM. The authors trained the classifier on data up until 2010 and test it against data from 2011 to 2014, the result of precision is 60% while the Flawfinder static analysis tool reaches only 1% at the same level of recall.

Grieco et al. [7] proposed an approach that utilize lightweight static and dynamic features to predict whether a binary program is likely to contain a software vulnerability. Both static features and dynamic features try to abstracted the use patterns of the C standard library, the difference is static features are the set of potential subsequences of function calls, while dynamic features are obtained by analyzing program execution traces containing concrete function calls augmented with its arguments. In the experiments the paper covered, the authors used Bag-of-Words and word2vec to process the different sets of features. To addressed the class imbalance issue, the paper used a well tested solution called random oversampling [13]. The authors trained several machine learning classifiers: logistic regression, MLP of single hidden layer and random forest. The result of this paper showed that the best performing model is a random forest trained with dynamic features, vectorized with 2- or 3- grams, achieving an average test error of 31%.

Li et al. [8] designed Vulnerability Deep Pecker, a deep learning-based vulnerability detection system. To address the problem that existing vulnerability detection system rely on human experts to define features and often incur high false negative, the paper first defined code gadgets which is a number of lines of code that are semantically related to each other, and used it to represent programs. The authors select Recurrent Neural Networks (RNN) model and use Long Short-Term Memory (LSTM) to address the Vanishing Gradient problem. Then the authors find Bidirectional LSTM is more suitable for the reason that the argument(s) of a program function call may be affected by earlier statements or the later statements. Vulnerability Deep Pecker has two phase: the learning phase and detection phase. In the learning phase, the system extract the library or API functions and the corresponding program slices, generate code gadgets and their corresponding labels, transform code gadgets to vector representation and train a BLSTM neural network. In the detection phase, the system first transform target programs into code gadgets and vectors, and then detected in the trained model. The authors collect 10791 programs related to vulnerabilities, and shows that Vulnerability Deep Pecker can achieve much lower false negative rate than other vulnerability detection system.

Lin et al. [9] proposed a data-driven method to address the shortage of high-quality training data and relying on the hand-crafted features in vulnerability discovery using machine learning [10]. First, the authors labeled 457 vulnerable functions and collected 32531 non vulnerable functions from six projects, and extracted the abstract syntax trees (ASTs) from source code using "CodeSensor", which is a robust parser implemented by [11]. Afterward, the authors converted the serialized ASTs to equal-length sequences while preserving the structural and semantic features. The paper proposed a BLSTM neural network, which takes the sequences above mentioned as input. The first layer of the network is word2vec embedding layer which maps each element of the sequence to a vector, and the second layer is an LSTM layer which contains 64 LSTM units in a bidirectional form. To accommodate large ASTs for extracting the latent sequential features in ASTs, the third layer of the network is a global max pooling layer. According to the reported results, the method of the paper proposed are more effective for predicting vulnerable functions, both within a project and across multiple projects compared with the traditional code metrics.

The following Table 1 gives a summary of recent works on static analysis using machine learning techniques.

Table 1. Summary of recent works on static analysis using machine learning techniques.

Paper	Approach summary	Advantages	Limitations	Future work
Scandariato et al. [4]	Bag-of-Words + Naive Bayes/Random Forest	Within-project	Cross-project	Expend to cross-project
Shin et al. [5]	Bidirectional models with RNN hidden units	Recognize functions of binaries code	Rely on training data	Explain the internal mechanics
Perl et al. [6]	Bag-of-Words + SVM	Precision and recall	Rely on the manual analysis before train	Minimize the likelihood
Grieco et al. [7]	Utilize static and dynamic features + Bag-of-Words/ Word2Vec + logistic regression/random forest	Accuracy	Test cases is small	Introduce 1D version of a CNN
Li et al. [8]	Code gadget + BLSTM of RNN	Lower false negative rate + not rely on hand-crafted features	Only contains buffer error and resource management error et al.	Solve the limitations
Lin et al. [9]	CodeSensor + BLSTM of RNN	More effective + not rely on training data and the hand-crafted features	Not apply to vulnerabilities involve multiple functions or files	Solve the limitations

2.3 Discussion

In the previous subsection, we reviewed and summarized several recent studies in the field of static analysis using machine learning techniques. A glance summary of all the articles reviewed in this section is also presented in Table 1. Some concluding points, challenges and possible future work could be drawn from the review of previous studies.

A statistical conclusion in the field of static analysis using machine learning techniques is the fact that Random Forest model can achieve the best result among the common used machine learning algorithm. However, as deep learning grow more and more mature, the technique has been used widely in this area and the results of the experiments using deep learning are more inspiring, especially the Bidirectional LSTM model of RNN algorithm. The features extracted from training data have to be processed before joined to the learning model, and the most widely used techniques are Bag-of-Words and word2vec.

Static analysis using machine learning techniques can find technical software vulnerabilities as well as logic vulnerabilities. In our opinion, this is a promising area, and more and more serious vulnerabilities will be discovered by this means. However, there are many limitations to static analysis using machine learning techniques. First, the approaches proposed by the papers reviewed earlier are not full-automatic and the results are unstable, because it rely on the quality of training data collected manual and the hand-crafted features selected by experts in a varying degrees. Second, all the methods the reviewed paper mentioned have to face the fact that the collected dataset suffers from a severe class imbalance. Last but not least, the experiments the reviewed paper implemented mostly aimed at source code, and the results to the binary code is not reasonable. However, the program we encountered in reality mostly have no source code.

It is clear that there is still room for much further progress in the field of static analysis using machine learning techniques. Some possible future works drawn from review of previous studies is as follows: introduce reinforcement learning to guide the learning of the training model and use transfer learning to address the problem of training data and test case vary significantly; combine static analysis with other vulnerability discover techniques, for instance, symbolic execution and fuzzing; conduct a learning model can achieve more reasonable result to binary code.

3 Symbolic Execution Using Machine Learning Techniques

3.1 Principles of Symbolic Execution

Symbolic execution [14] is a method for program reasoning that uses symbolic values as inputs instead of actual data, and it represents the values of program variables as symbolic expressions on the input symbolic values. No matter when a judgment and jump statement is encountered, the method will put the path constraints of the current execution path into the constraint set of the path. The path constraint refers to the value of the branch condition related to the input symbol and the path constraint set is used to store the constraints collected on each program path. We can obtain the accessibility of the path by constraint solver. If the result of the constraint solving has a solution, it means that the path is reachable, otherwise it means that the path is unreachable. In the ideal case of sufficient time and computing resources, the symbol execution can traverse all the paths of the target program and judge its accessibility.

3.2 Summary of Recent Works

Li et al. [15] utilize machine learning to address the major obstacle in applying symbolic execution to real world programs. The problem is the capability of constraint solving, which is closely related with the optimization problem: finding solutions to minimize the dissatisfaction degree. Unlike concolic testing and heuristic search, the authors proposed MLB, a new symbolic execution tool, driven by Machine Learning Based constraint solving. MLB encodes all the difficult operations as symbolic constraints and transforms the feasibility problems of the path conditions into optimization problems. Here, the paper adopt a machine learning based optimization method named RACOS (randomized coordinate shrinking) classification algorithm [16], which learns to discriminate good and bad solutions while trying to keep the error-target dependence and the shrinking rate small. According to the reported results, the method of the paper proposed are more effective and efficient with the instruction coverage reach to 89% and the instruction efficiency reach to 0.44%/s.

Meng et al. [17] proposed a new method combined symbolic execution with machine learning technique to discover vulnerability. Firstly, the authors collect vulnerable functions from CVE and NVD. Then they dig similar function set which have the most features in common in code base with the defined cosine distance. Secondly, vulnerable function call graphs can be extracted from source code base and it can be used to guide the symbolic execution engine to reach the target function. Finally, path constraint can be calculate through the constraint solver and then estimate the sink point according to vulnerability checking rules. The result of this paper showed that symbolic execution utilize machine learning can reach the vulnerable function of FFmpeg within 36 s while symbolic execution only need 8 h.

3.3 Discussion

In the previous subsection, we reviewed and summarized recent work in the field of symbolic execution using machine learning techniques. Symbolic execution is a promising approach to be used in vulnerability discovery, but previous works in this area suffer from some important limitations. In our opinion, path explosion and constrain solve are the two main challenges and machine learning techniques are introduced to address them in recent years. Researchers always concentrate on getting more valuable paths or optimizing constraint solver using machine learning techniques. The result is not inspiring and more efforts are needed in the future.

4 Fuzzing Using Machine Learning Techniques

4.1 Principles of Fuzzing

Fuzzing [18] is a highly automated testing technique that covers numerous boundary cases using invalid data (from files, network protocols, API calls, and other targets) as application input to better ensure the absence of exploitable vulnerabilities. It contains three aspects [19]: firstly, it generates semi-valid or random data; secondly, it sends the generated data into the target application; finally, it observes the application to see if it fails as it consumes the data. Semi-valid data is data that is correct enough to pass input examinations, but still invalid enough to cause problems.

The quality of the Semi-valid data is one of the most important factors that influence the effectiveness and efficiency of fuzzers. There are two main methods of data generation, including data-generation technique and data-mutation technique. Data-generation technique is usually based on specifications, such as file format specifications and network protocol specifications, to generate data. Data-mutation technique generates data by modifying some fields of valid inputs. When specifications are very complex, data-mutation is more appropriate, but the code coverage rate may be very low. So researchers introduced machine learning techniques to address the problems in this area.

4.2 Summary of Recent Works

Böhme et al. [20] model the American Fuzzy Lop (AFL) as a systematic exploration of the state space of a Markov chain and take the probability that fuzzing a seed which exercises program path i generates a seed which exercises path j as transition probability p_{ij}. The paper also improve the power schedules by assigning energy that is inversely proportional to the density of the stationary distribution. To fuzz the best seeds early on, the authors introduce a different search strategy that chooses seeds earlier exercise lower frequency paths and have been chosen less often. As evidenced by the experiments the paper mentioned, the method can exposes an order of magnitude more unique crashes than AFL in the same time budget.

Godefroid et al. [21] introduced neural-network-based learning techniques to fuzzing field and proposed Samplefuzz algorithm which leverages a learnt input probability distribution in order to intelligently guide where to fuzz well-formed inputs. The paper first presents an overview of the PDF format and pick up the "Objects" as the mutate element. To learn a generative model of PDF objects, the authors consider PDF objects as a sequence of characters and use a recurrent neural network based character-level language model (char-rnn). After the learnt char-rnn model has been created, the paper adopted SampleSpace as the sampling strategy. At last, the paper use Samplefuzz algorithm to create new PDF object instances, but at the same time introduce anomalies to exercise error-handling code.

Wang et al. [22] propose Skyfire, a novel data-driven seed generation approach, to improve the performance of the American Fuzzy Lop (AFL) fuzz testing framework. Firstly, the authors collect a vast amount of samples and abstract syntax trees (ASTs) based on the grammar. Secondly, Skyfire learns a probabilistic context-sensitive grammar (PCSG), which describes both syntax features and semantic rules for highly-structured inputs. Then, Skyfire leverage PCSG to generate seed inputs by iteratively selecting and applying a production rule on a non-terminal symbol until there is no non-terminal symbol in the resulting string. Finally, Skyfire randomly replace a leaf-level node in the AST with the same type of nodes to mutates the remaining seed inputs. According to the reported results, the method of the paper proposed can effectively improve the code coverage of fuzzers can significantly improve the capability of fuzzers to find bugs.

Nichols et al. [23] propose to use Generative Adversarial Network (GAN) models and Long Short Term Memory (LSTM) to increase the rate of unique code path discovery of AFL. Firstly, the authors run AFL on a target program for a fixed amount of time to produce the training data. As for the LSTM model, the authors use a 128 wide initial layer, an internal dense layer, a final softmax activation layer and a categorical cross-entropy loss function. The model takes in a seed sequence sampled from the training corpus and predicts the next character in the sequence. Then, the GAN [24] architecture the paper mentioned has two models. The generative model G is a fully connected 2 layer DNN with a ReLU non-linearity to generate realistic output and the discriminative model D is a 3 layer DNN to predict if the data is true or fake. The result of this paper showed that GAN was faster and more effective than the LSTM, and GAN helps AFL discover 14.23% more code paths, finds 6.16% more unique code paths, and finds paths that are on average 13.84% longer.

The following Table 2 gives a summary of recent works on fuzzing using machine learning techniques.

Table 2. Summary of recent works on fuzzing using machine learning techniques.

Paper	Approach summary	Use AFL	Future work
Böhme et. al. [20]	Markov chain	Yes	
Godefroid et al. [21]	char-rnn	No	Reinforcement learning
Wang et al. [22]	PCSG	Yes	Extend the method to more languages and complier
Nichols et al. [23]	GAN+LSTM	Yes	Reinforcement learning

4.3 Discussion

In the previous subsection, we reviewed and summarized several recent studies in the field of fuzzing using machine learning techniques. A glance summary of all the articles reviewed in this section is also presented in Table 2, where we have specified the key differentiating factors of each work.

In the field of fuzzing, machine learning techniques are introduced to guide the process of the input tests generating and mutating. Many researchers tend to extend AFL, the state-of-the-art coverage-based fuzzers, and adopted RNN, Markov, GAN and some algorithms of NLP to improve the performance of AFL. According to the papers, the result is inspiring, but there is still room for much further progress. Some possible future works drawn from review of previous studies is reinforcement learning.

5 Conclusion

Machine learning techniques have been successfully used in the domain of software vulnerability discovery. In this paper, we extensively reviewed previous work and organized the studies in three main categories. For each category, we provided a short yet sufficiently detailed summary of each work and discussed the concluding points, challenges and possible future work.

Acknowledgement. This work was supported by National Key Research & Development Plan of China under Grant 2016QY05X1000, National Natural Science Foundation of China under Grant No. 61771166, and Dongguan Innovative Research Team Program under Grant No. 201636000100038.

References

1. Nayak, K., Marino, D., Efstathopoulos, P., Dumitraş, T.: Some vulnerabilities are different than others. In: Stavrou, A., Bos, H., Portokalidis, G. (eds.) RAID 2014. LNCS, vol. 8688, pp. 426–446. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11379-1_21
2. Chen, Q.: Bridges, R.: Automated behavioral analysis of malware: a case study of WannaCry Ransomware. In: the 16th IEEE International Conference On Machine Learning And Applications, pp. 454–460, Cancun, Mexico (2017). https://dblp.uni-trier.de/pers/hd/c/Chen:Qian
3. Liu, B., Shi, L., Cai, Z., Li, M.: Software vulnerability discovery techniques: a survey. In: the 4th International Conference on Multimedia Information Networking and Security, Nanjing, China (2012)
4. Scandariato, R., Walden, J., Hovsepyan, A., Joosen, W.: Predicting vulnerable software components via text mining. IEEE Trans. Softw. Eng. **40**(10), 993–1006 (2014). https://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=32
5. Shin, E., Song, D., Moazzezi, R.: Recognizing functions in binaries with neural network. In: the 24th USENIX Security Symposium, Washington, D.C., USA (2015)
6. Perl, H., Dechand, S., Smith, M.: VCCFinder: finding potential vulnerabilities in open-source projects to assist code audits. In: Proceeding of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 426–437, Denver, Colorado, USA (2015)
7. Grieco, G., Grinblat, G., Uzal, L., Rawat, S., Feist, J., Mounier, L.: Toward large-scale vulnerability discovery using machine learning. In: Proceedings of the 6th ACM Conference on Data and Application Security and Privacy, pp. 85–96, San Antonio, TX, USA (2015)
8. Li, Z.: VulDeePecker: a deep learning-based system for vulnerability detection. In: the 25th Annual Network and Distributed System Security Symposium, NDSS, San Diego, California, USA (2018)
9. Lin, G., Zhang, J.: Cross-project transfer representation learning for vulnerable function discovery. IEEE Trans. Ind. Inf. **14**, 3289–3297 (2018). https://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=9424
10. Chen, L., Yang, C., Liu, F., Gong, D., Ding, S.: Automatic mining of security-sensitive functions from source code. CMC: Comput. Mater. Cont. **56**(2), 199–210 (2018)
11. Yamaguchi, F., Lottmann, M., Rieck, K.: Generalized vulnerability extrapolation using abstract syntax trees. In: Proceedings of the 28th Annual Computer Security Applications Conference, pp. 359–368 (2012)

12. Ghaffarian, S., Shahriari, H.: Software vulnerability analysis and discovery using machine-learning and data-mining techniques: a survey. ACM Comput. Surv. **50**(4) (2017)
13. He, H., Garcia, E.: Learning from imbalanced data. IEEE Trans. Knowl. Data Eng. **21**(9) (2009)
14. Chu, D.H., Jaffar, J., Murali, V.: Lazy symbolic execution for enhanced learning. In: the 5th International Conference on Runtime Verification, pp. 323–339, Toronto, ON, Canada (2014). https://link.springer.com/conference/rv
15. Li, X.: Symbolic execution of complex program driven by machine learning based constraint solving. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pp. 554–559, Singapore, Singapore (2016)
16. Yu, Y., Qian, H., Hu, Y.Q.: Derivative-free optimization via classification. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, pp. 2286–2292 (2016)
17. Meng, Q., Wen, S., Zhang, B., Tang, C.: Automatically discover vulnerability through similar functions. In: 2016 Progress in Electromagnetic Research Symposium (PIERS), Shanghai, China (2016). https://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7655139
18. Oehlert, P.: Violating assumptions with fuzzing. IEEE Secur. Priv. **3**(2), 58–62 (2005)
19. Liu, B., Shi, L., Cai, Z., Li, M.: Software vulnerability discovery techniques: a survey. In: Fourth International Conference on Multimedia Information Networking and Security (2012)
20. Böhme, M., Pham, V.T., Roychoudhury, A.: Coverage based greybox fuzzing as Markov Chain. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, NY, USA (2016)
21. Godefroid, P., Peleg, H., Singh, R.: Learn&Fuzz: machine learning for input fuzzing. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, pp. 50–59. Urbana-Champaign, IL, USA (2017)
22. Wang, J., Chen, B., Wei, L., Liu, Y.: Skyfire: data-driven seed generation for fuzzing. In: 2017 IEEE Symposium on Security and Privacy, San Jose, CA, USA (2017). https://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7957740
23. Nichols, N., Raugas, M., Jasper, R., Hilliard, N.: Faster fuzzing: reinitialization with deep neural models. arXiv preprint arXiv:1711.02807 (2017)
24. Li, C., Jiang, Y., Cheslyar, M.: Embedding image through generated intermediate medium using deep convolutional generative adversarial network. CMC: Comput. Mater. Con. **56**(2), 313–324 (2018)