2018 Eighth International Conference on Instrumentation and Measurement, Computer, Communication and Control

Current and Future Research of Machine Learning based Vulnerability Detection

[1, 2]Zhaoyan Jin [2]Yang Yu

1. Beijing University of Posts and Telecommunications, Beijing, China
2. Beijing Leadsec Technology Co.Ltd, Beijing, China

jinzhaoyan@163.com

***Abstract*—The quantity and volume of software increase rapidly, more vulnerabilities are hidden there, and thus vulnerability detection becomes more important. This paper reviews works about machine learning based vulnerability detection methods for source code, especially on program representation and vectorization, and Machine Learning based Methods. However, current machine learning methods detect vulnerabilities in coarse-grained level, and locating the vulnerabilities still needs much additional work. This problem can be solved by vulnerability detection in fine-grained level. This paper assumes that vulnerability detection in fine-grained level will be a future research trend, and proposes several possible solutions.**

Keywords—vulnerability detection, machine learning, abstract syntax tree, neural network

I. INTRODUCTION

In computing security, vulnerability is a flaw or weakness in program, which can be exploited by attackers to violate the confidentiality, integrity and availability of computing systems. Because of programmers' carelessness or lack of secure coding consciousness, there are more or less vulnerabilities in operating system kernels and all kinds of applications. One popular example is the "Heartbleed" vulnerability found in April 2014 in the cryptographic library OpenSSL, which provides encrypted transmission of Web pages on many systems. By exploiting a missing-sanity-check vulnerability, attackers can read sensitive information from the websites serving encrypted pages.

Traditional vulnerability discovery methods include the static analysis of code and the dynamic testing of program. The static analysis method, such as taint-analysis [1] and symbolic execution [2], analyzes the source code of program, and utilizes the rules summarized by security experts to discover vulnerabilities [3]. In the static analysis based vulnerability discovery method, the hugest challenge is the exponential explosion of code branch paths that need to be covered. The dynamic testing method, such as fuzzing, executes the program by providing some inputs, and observes the executing status and return value to decide if some vulnerability exists. In order to guide the mutation of inputs and find an input crashing the program as soon as possible, dynamic testing usually combines with the static analysis method. For example, the most popular concolic execution [4] is the combination of fuzzing and symbolic execution. The above methods only detect a limited subset of possible vulnerabilities based on pre-defined rules. Moreover, they focus on discovering vulnerabilities in a file or a function, and while be faced with massive source codes, they seem to be inefficient.

Machine learning based vulnerability detection methods [5] detect whether or not some vulnerability exists in a code segment, such as a document, a function or a gadget. These methods can encode vulnerability information into some model while training with massive source codes, and detect similar vulnerabilities in additional source code with the trained model. The same as other machine learning tasks, most of the time used in vulnerability detection is in the training step, and the time used in the detection step is trivial. In order to make the machine learning method to work, source codes or programs must be represented reasonably, embedded into a vector space, and then used as the input of the learning methods. Current machine learning based vulnerability detection methods predict vulnerability in coarse-grained level, such as modular, document, function or gadget, and additional works, such as fuzzing, must be applied in order to locating the predicted vulnerability. Machine learning based methods that can both detect and locate vulnerability will be a future research trend.

This paper is organized as follows. Section 2 introduces the representation and Vectorization techniques of programs. Section 3 reviews related machine learning based methods used in vulnerability detection. Future research and conclusion are given in section 4 and 5.

II. PROGRAM REPRESENTATION AND VECTORIZATION

In order to represent a program and embed it into a vector space, this paper assumes that the source code of a program is known. If the source code is unknown, we can disassemble a binary executable into assemble language, convert the assemble language into some intermediate representation, and then use the intermediate representation as a kind of source code [6]. The conversion of binary executable into its source code is beyond the scope of this paper.

Before source code of a program is embedded into a vector space, it must be represented with some reasonable metrics or semantics. This paper describes four metrics or semantics that are commonly used to represent source code: software metric, language model, tree representation and graph representation.

978-1-5386-8246-3/18/$31.00 ©2018 IEEE
DOI 10.1109/IMCCC.2018.00322

A. Software Metric

Software development is a complex process, and software metric characteristics this process qualitatively and quantitatively. Metric of software development process can be quantified by complexity, code churn and developer activity metrics, and these metrics can be applied to identify vulnerabilities [7]. With the reasonable hypothesis that code with recent changes (i.e. code churn) might have new vulnerabilities, and files worked by multi developers might have more vulnerabilities than worked by single developer, software metric can characteristic the vulnerability information.

SHIN [8] makes a comprehensive study on the relation between software metrics and vulnerabilities. His results indicate that the software metrics are discriminative and predictive of vulnerabilities, and the experiments on the Mozilla Firefox and the Linux kernel show that machine learning is a promising method to detect vulnerabilities. However, software metrics are coarse-grained representation of code, and the detection capability is limited compared with other semantics.

B. Language Model

Programming language is a formal language, and production rules of context-free grammar are used to generate source code of program. This process is similar to natural language, so the language model in natural language process can be used to analyzing source code.

Bag-of-words model is a basic model to process natural language. In the thesis [3], Yamaguchi introduces three bag-of-words models to represent features of source code: token-based, symbol-based, and multi-stage feature map. The token-based feature map considers each token in source code to be a word, counts the number of occurrences for each word, and uses these numbers of occurrences as a vector. Instead of token, the symbol-based feature map considers each symbol to be a word, and uses the occurrences of each symbol as an element of a vector. Here, a symbol, such as "struct bar *", is a basic grammar element, and may contain several tokens. The multi-stage feature map first maps source code to a vector, then groups all words into clusters, and finally uses the occurrences of each cluster as the element of a vector. Here, the dimension of the vector is equal to the number of clusters, and the words are clustered with their similarities. For example, the words of "malloc" and "calloc" are grouped into the same cluster.

Li et al. [9] have proposed using code gadgets to represent program source code, and then embed them into a vector space, where a code gadget is a number of lines of semantically related code. The total process is as follows: firstly, the backward function call analysis is applied to generate gadgets; secondly, function names and variables are substituted with predefined names to reduce the dimension of tokens; thirdly, each token is embedded into a vector space via word2vec [10] and a gadget is the concatation of its vector sequences; and finally, Bidirectional Long Short-Term Memory neural network is used as the prediction model.

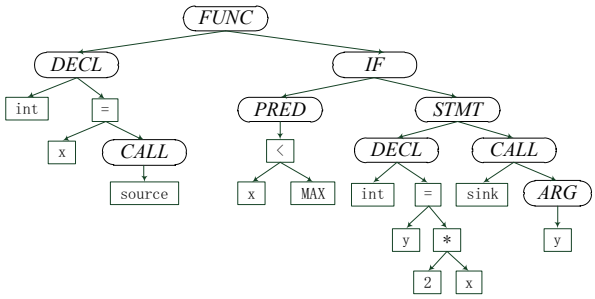
C. Tree Representation

The source code of a program is generated from context-free grammar, and thus can be parsed as a concrete syntax tree or parser tree. Based on the concrete syntax tree, abstract syntax tree can be acquired. The abstract syntax tree contains the semantic of the program, and is usually applied to analyze the source code statically. Figure 1 is a piece of source code, and figure 2 illustrate its corresponding abstract syntax tree [11]. While analyzing the source code statically, dominator tree, post-dominator tree, tainted execution path tree and other tree representations can be acquired from the abstract syntax tree.

Fig. 1. Sample of source code

```
1. void foo()
2. {
3.     int x = source();
4.     if (x < MAX)
5.     {
6.         int y = 2 * x
7.         sink(y)
8.     }
9. }
```

Fig. 2. Abstract syntax tree of the source code



In order to input the tree representations into a machine learning system for training, they must be embedded into a vector space, and this embedding process is called vectorization. Researchers have developed many methods for embedding a tree representation into a vector space. Alon et al. [12, 13] have proposed a path-based embedding method. In their method, two leaf nodes a and b use their first common ancestor c as an intermediate node, the path between a and c is $a\uparrow\cdots\uparrow c$, the path between c and b is $c\downarrow\cdots\downarrow b$, the path between a and b is $a\uparrow\cdots\uparrow c\downarrow\cdots\downarrow b$, and a tree is represented as the paths of all leaf node pairs. Pradel and Sen [14] have proposed an abstract syntax tree based context to represent a tree structure. In their method, a *str* function is first introduced to get the string value of each node; then, each string value is coded as a one-hot vector; next, the abstract syntax tree context is represented as the concatenation of an eight-tuple; and finally a node and its context are considered to be a pair to train a neural network. In addition, Medeiros et al. [1] have proposed to detect and remove input validation vulnerabilities, such as SQL injection vulnerabilities, with the tainted execution path tree. In their method, the tainted execution path tree is applied to detect candidate vulnerabilities, and machine learning model is applied further to reduce the false positive rate. In their conclusion, string manipulation, validation, and SQL query manipulation are three main sets of

attributes that lead to false positives, and they are applied to represent and embed source code.

D. Graph Representation

Besides the tree representation, graph representation can also be constructed from the abstract syntax tree. Graph representation is more expressive and contains more semantics of source code than the tree representation. The graph representations, such as control flow graph [3, 15], program dependence graph [3], code property graph [3] and program graph [16], capture the semantics from different viewpoints.

In the basic control flow graph, each statement is a node, and the control flow between two statements constructs an edge [3]. Harer et al. [15] apply the function level control flow graph to capture the semantic of source code. In this kind of control flow graph, each node is a basic code block with no branching operation, each edge describes the control flow between two nodes during execution, and the control flow graph is a graph representation of the different paths that a program can take during execution.

Instead of describing the control follow of source code explicitly, program dependence graph can also be applied to capture the control follow semantic of source code. The program dependence graph [17] contains two types of dependencies derivable from control flow, data and control dependencies. Based on tree representation and graph representation, Yamaguchi has developed the code property graph [3], which is the union of abstract syntax tree, control flow graph, program dependence graph, dominator tree and post-dominator tree. The code property graph is a joint representation of syntax, control flow and data flow, and it can also include any other graph representation parsed from the abstract syntax tree.

Allamanis et al. [16] have proposed the concept of program graph to capture the syntactic and semantic relationships of program source code. Again, the backbone of a program graph is the abstract syntax tree of the program. In an abstract syntax tree, syntax nodes correspond to non-terminals in the programming language's grammar, and syntax tokens correspond to terminals. Additional edges are added to an abstract syntax tree, and different edge types are applied to capture the control flow and data flow of a program.

In [3], Yamaguchi has proposed the graph-based feature map, which is a general method for embedding graph representation (including tree representation) into a vector space. The main idea is to enumerate all substructures of a graph and use them as dimensions of the feature space. Further, in order to speed up this embedding process, a hash value is calculated for each substructure, so the vector space is now a set of hash values.

III. Machine Learning Based Methods

After representing and embedding source code into vector space, machine learning methods can be used to model source code and detect vulnerabilities. According to whether or not labeled vulnerabilities are given, vulnerability detection methods based on machine learning can be classified into supervised and unsupervised learning.

A. Supervised Learning

Although the United States National Vulnerability Database (NVD) provides much vulnerability information, it is very hard to match vulnerability information with its source code. While analyzing the source code repository for detecting vulnerabilities, labeled data are very hard to acquire.

VulDeePecker [9] is the first dataset for evaluating deep learning-based vulnerability detection systems, which contains 61,638 code gadgets, including 17,725 vulnerable code gadgets and 43,913 benign code gadgets. Harer et al. [15] compile a large dataset of hundreds of thousands of open-source functions labeled with the outputs of a static analyzer. Pradel and Sen [14] extract positive training examples from a code repository, apply simple program transformations to create negative training examples, and their proposed DeepBugs framework is trained to find bugs. Manual analysis is a common way to get labeled training data. Medeiros et al. [1] manually extract features of input validation vulnerabilities, and use this features to reduce false positive.

Many supervised learning methods have been applied to detect vulnerabilities, such as Naïve Bayes [1], Support Vector Machine [1], random forests [15], convolutional neural network [15], sequence models [18], Bidirectional Long Short-Term Memory neural network [9], and so on. However, [15] and [9] show that convolutional neural network and Bidirectional Long Short-Term Memory neural network outperform other supervised learning methods while detecting vulnerabilities.

B. Unsupervised Learning

Unsupervised learning methods apply the patterns or rules summarized by security experts to discover vulnerabilities [1, 19-21].

In [19], dimensionality reduction is applied to represent the patterns of code repository. Firstly, abstract syntax trees are extracted from code repository; secondly, every abstract syntax tree is embedded into vector space, and a matrix is constructed; thirdly, dimensionality reduction techniques are applied to the matrix; and finally, given a known vulnerability, vectors with similar patterns are retrieved as candidate vulnerabilities.

In [20], the code property graph is extended to be a definition graph, which takes the relationships between functions into consideration. Simple recursive procedure is applied to enumerate all components of the definition graph, and then API symbols and invocations are clustered. Finally, template for taint-style vulnerability as a graph traversal in the query language Gremlin is constructed, and is applied to retrieved taint-style vulnerabilities.

In [21], anomaly detection is applied to detection vulnerabilities. Firstly, sinks and sources are identified in the source code; secondly, functions with similar context are grouped; thirdly, the tainted variables depending on the sinks and sources are identified; fourthly, functions are embedded into a vector space based on the tainted conditions; and finally,

function with missing validation conditions are detected as candidate vulnerabilities.

In [1], taint-analysis is also applied to detect input validation vulnerabilities. In the work, the authors analyze the vulnerabilities manually, and find that string manipulation, validation and SQL query manipulation are three main sets of attributes that lead to false positives.

IV. Future Research

Related works show that, machine learning, especially supervised learning if labeled data is given, is an effective method to detect vulnerabilities. Nowadays, these methods detect vulnerabilities for a document, a function, a gadget, or a snippet, and they are coarse-grained and hard to locate the identified vulnerabilities. So, machine learning based methods, those can both detect and locate vulnerabilities, will be a future research trend.

This section presents some possible methods that can both detect and locate vulnerabilities. Here, labeled data with locations of vulnerabilities is assumed to be known. If some vulnerability is known in some source code, but the location is unknown, concolic execution [4] can be applied to locate the vulnerability. To acquire an available dataset, NVD and other vulnerability datasets can be used.

In a program of source code, a vulnerability is defined as a line of code that the program crashes there, and if each line of code is sure to be a vulnerability or not, then hidden Markov model can be applied to model the source code. In the hidden Markov model, each line of code is an observation, its hidden state implies whether or not the code is vulnerability, and the relationships between hidden states can be modeled by a Markov process.

The capability of hidden Markov model is limited, and it models the source code of a program as a sequence. However, the typical semantic representation of source code is its abstract syntax tree. So, a natural extension of the hidden Markov model is the recursive or recurrent neural network, which has been applied to many program analyzing tasks, such as program completion [16] and program translation [22].

Moreover, if the path from the entry to vulnerability is a vulnerability path, the path from the entry to a benign line of code is a benign path, and then a program is modeled to be a search tree. In the search tree, only some leaf nodes are vulnerability nodes, and others are benign nodes. Vulnerability detection and location in the search tree is converted to find those vulnerability paths, and thus the reinforcement learning method can be used to solve the problem.

V. Conclusion

With the rapid increase of software quantity and volume, more and more vulnerabilities will be hidden in software, and thus vulnerability detection becomes more and more important. Machine learning based vulnerability detection methods can help developers find vulnerabilities before releasing it on the market. This paper introduces four kinds of method to represent and embed source code into a vector space, and they are software metric, language model, tree representation and graph representation. Then, the paper reviews some machine learning methods that have been used in detecting vulnerabilities. However, current machine learning based vulnerability detection methods identify and locate vulnerabilities in a coarse-grained level, and other expensive locating methods are needed. The paper assumes that vulnerability detection in fine-grained level will be a future research trend, and proposes several possible solutions.

References

[1] Medeiros, I.; Neves, N. & Correia, M. Detecting and removing web application vulnerabilities with static analysis and data mining IEEE Transactions on Reliability, IEEE, 2016, 65, 54-69.

[2] Ayewah, N.; Hovemeyer, D.; Morgenthaler, J. D.; Penix, J. & Pugh, W. Using static analysis to find bugs IEEE software, IEEE, 2008, 25.

[3] Yamaguchi, F. Pattern-Based Vulnerability Discovery 2015.

[4] Stephens, N.; Grosen, J.; Salls, C.; Dutcher, A.; Wang, R.; Corbetta, J.; Shoshitaishvili, Y.; Kruegel, C. & Vigna, G. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. NDSS, 2016, 16, 1-16.

[5] Ghaffarian, S. M. & Shahriari, H. R. Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey ACM Computing Surveys (CSUR), ACM, 2017, 50, 56.

[6] Wang, R.; Shoshitaishvili, Y.; Bianchi, A.; Machiry, A.; Grosen, J.; Grosen, P.; Kruegel, C. & Vigna, G. Ramblr: Making Reassembly Great Again Proceedings of the 24th Annual Symposium on Network and Distributed System Security (NDSS'17), 2017.

[7] Jimenez, M.; Papadakis, M. & Le Traon, Y. Vulnerability prediction models: a case study on the linux kernel Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on, 2016, 1-10.

[8] Shin, Y.; Meneely, A.; Williams, L. & Osborne, J. A. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities IEEE Transactions on Software Engineering, IEEE, 2011, 37, 772-787.

[9] Li, Z.; Zou, D.; Xu, S.; Ou, X.; Jin, H.; Wang, S.; Deng, Z. & Zhong, Y. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection arXiv preprint arXiv:1801.01681, 2018.

[10] Mikolov, T.; Chen, K.; Corrado, G. & Dean, J. Efficient estimation of word representations in vector space arXiv preprint arXiv:1301.3781, 2013.

[11] Yamaguchi, F.; Golde, N.; Arp, D. & Rieck, K. Modeling and discovering vulnerabilities with code property graphs Security and Privacy (SP), 2014 IEEE Symposium on, 2014, 590-604.

[12] Alon, U.; Zilberstein, M.; Levy, O. & Yahav, E. A general path-based representation for predicting program properties Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2018, 404-419.

[13] Alon, U.; Zilberstein, M.; Levy, O. & Yahav, E. code2vec: Learning Distributed Representations of Code arXiv preprint arXiv:1803.09473, 2018.

[14] Pradel, M. & Sen, K. Deep Learning to Find Bugs 2017.

[15] Harer, J. A.; Kim, L. Y.; Russell, R. L.; Ozdemir, O.; Kosta, L. R.; Rangamani, A.; Hamilton, L. H.; Centeno, G. I.; Key, J. R.; Ellingwood, P. M. & others Automated software vulnerability detection with machine learning arXiv preprint arXiv:1803.04497, 2018.

[16] Allamanis, M.; Brockschmidt, M. & Khademi, M. Learning to Represent Programs with Graphs arXiv preprint arXiv:1711.00740, 2017.

[17] Ferrante, J.; Ottenstein, K. J. & Warren, J. D. The program dependence graph and its use in optimization ACM Transactions on Programming Languages and Systems (TOPLAS), ACM, 1987, 9, 319-349.

[18] de Sousa Medeiros, I. V. Detection of Vulnerabilities and Automatic Protection for Web Applications UNIVERSIDADE DE LISBOA, UNIVERSIDADE DE LISBOA, 2016.

[19] Yamaguchi, F.; Lottmann, M. & Rieck, K. Generalized vulnerability extrapolation using abstract syntax trees Proceedings of the 28th Annual Computer Security Applications Conference on, 2012, 359-368.

[20] Yamaguchi, F.; Maier, A.; Gascon, H. & Rieck, K. Automatic Inference of Search Patterns for Taint-Style Vulnerabilities 2015 IEEE Symposium on Security and Privacy, 2015, 797-812.

[21] Yamaguchi, F.; Wressnegger, C.; Gascon, H. & Rieck, K. Chucky: exposing missing checks in source code for vulnerability discovery Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, 2013, 499-510.

[22] Chen, X.; Liu, C. & Song, D. Tree-to-tree Neural Networks for Program Translation international conference on learning representations, 2018.