Learning Binary Code with Deep Learning to Detect Software Weakness

Young Jun Lee[*], Sang-Hoon Choi[*], Chulwoo Kim[**], Seung-Ho Lim[***], Ki-Woong Park[*†]
[*]Department of Computer and Information Security, Sejong University
Seoul, South Korea
[**]Department of Computer Science, Pace University
New York, United State
[***]Division of Computer and Electronic Systems Engineering, Hankuk University of Foreign Studies
Yongin, South Koreaa
[e-mail: firmcode@naver.com, csh0052@gmail.com, ck90134n@pace.edu, slim@hufs.ac.kr, woongbak@sejong.ac.kr]

Abstract

As more software are being developed, the importance of automated vulnerability analysis tools is increasing. In this paper, we propose a method of learning assembly code using deep learning to find software weaknesses. Unlike prior studies based on API function call sequence, our method starts by adding the assembly code to an immutable vector to learn the assembly language through deep learning. When modeling assembly code, we propose *Instruction2vec*, which is effective in vectorizing the assembly code. After learning the assembly code of the existing functions through the vector created by *Instruction2vec*, we classify whether the new functions have software weaknesses or not. In order to demonstrate the effectiveness of *Instruction2vec*, we train the vectors with text-convolutional neural network model (Text-CNN) and compare the results with those of *Word2vec*. We also used the *Juliet* test suite from National Institute of Standards and Technology as a dataset. As a result, we identify up to 96.1% accuracy in classifying whether the function has software weaknesses.

***Keywords*:** Deep Learning, Software Weakness, Convolutional Neural Network (CNN)

1. Introduction

As more software are being developed, the importance of automated vulnerability analysis tools is increasing. Typically, a static analyzer analyzes code without execution and finds vulnerabilities based on patterns of the code. It enables you to analyze the entire code quickly with little resources. However, the limitation of this method is that it applies only to well-known programming patterns. Recently, automated vulnerability analysis tools have been combined with machine learning to overcome pattern-based limitations. Machine learning has improved accuracy by training a lot of data to classify it, unlike the prior method of matching preset vulnerable code patterns. In order to use machine learning, data preprocessing is necessary. This is done by vectorizing application programming interface (API) functions. Although this method has contributed to higher accuracy than matching preset vulnerable code patterns, it has limitations in finding vulnerabilities other than those based on API functions. Therefore, in this paper, we propose a method to preprocess the assembly code to find more vulnerabilities using *Instruction2vec* which vectorizes assembly language efficiently. In addition, we use a

† : corresponding author

This work was supported by the National Research Foundation of Korea (NRF) grant (NRF-2017R1C1B2003957, NRF-2016R1A4A1011761)

text-convolutional neural network (Text-CNN)[1] instead of classical machine learning models to improve accuracy. The Text-CNN is a deep neural network model that classifies the text data through trained class-labeled sentences. We use this approach for code classification using vectorized assembly code via *Instruction2vec*. We discuss how this new method can find software weaknesses compared to prior methods.

2. Background

2.1 Vulnerability discovery using Machine Learning

In 2011, F. Yamaguchi published a study on vulnerability using machine learning in "Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities Using Machine Learning"[2]. In the paper, Yamaguchi identified vulnerabilities by using a feature based on API usage pattern vectors. However, the approach has limitations that it is white-box based, and only API functions are included in the feature. G. Grieco has introduced a new approach in "Toward large-scale vulnerability discovery using machine learning"[3] in 2016. The method extracted static and dynamic features from a black-box to improve the limitation of the previous approach. Furthermore, when extracting features, it used *Word2vec*[4] and a bag of words, to vectorize API functions and argument values. Although this method has improved by extracting dynamic features, it still has significant limitations because only features based on API functions and argument values are used. In this paper, we propose a method to find software weaknesses by extracting features from assembly code, and using deep learning to overcome the limitations mentioned above. We can expect reasonably higher accuracy because the assembly code describes the structure of the program in greater detail than API functions.

2.2 Deep learning for classification

Convolutional neural networks[5] (CNN) is one of the deep learning algorithms and has a very high accuracy in image classification problems. Generally, it consists of an input, an output layer and multiple hidden layers having convolutional, pooling or fully connected layer. Convolutional layers apply a convolution operation to the input, having learnable filters. The operation slides each filter across the width and height of the input volume, and computes dot products between the entries of the filter and the input at any position. Unlike fully connected feedforward neural networks which generate a high number of neurons for shallow architecture, it reduces the number of free parameters, allowing the network to be deeper with fewer parameters. Pooling layers operate the outputs of neuron clusters at one layer into a single neuron in the next layer, using the max or average operation. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. Fully connected layers connect every neuron in one layer to every neuron in another layer, as seen in the multi-layer perceptron neural network. All activations in the previous layer can be computed with a matrix multiplication followed by a bias offset. Recently, CNN has been used in natural language processing (NLP) beyond the field of image recognition. The text-convolutional neural network model (Text-CNN) trains a CNN with one layer of convolution on top of word vectors obtained from an unsupervised neural language model. Words vectors are essentially feature-extractors that encode semantic features of words in their dimensions. In the present work, we use the Text-CNN that has fine-tuning hyperparameters for our task. Despite little tuning of hyperparameters, this model achieves an excellent result on software weakness discovery.

3. Methods

In this section, we describe overall architecture of our method to apply instruction vectors of a function instead of word vectors of a sentence as features on the Text-CNN. Learning instruction vectors result in further improvements. As depicted in **Fig. 1**, the model consists of preprocessing assembly code with the *Instruction2vec*, training the data through the Text-CNN, and finding software weaknesses through classification. The most important purpose of our method is to classify whether the assembly code is vulnerable or not.

First, in section 3.1, we compare the process of *Word2vec* and *Instruction2vec*. In section 3.2, we explain the principle of *Instruction2vec* in detail. Finally, in section 3.3, we describe the process by which preprocessed data is classified.

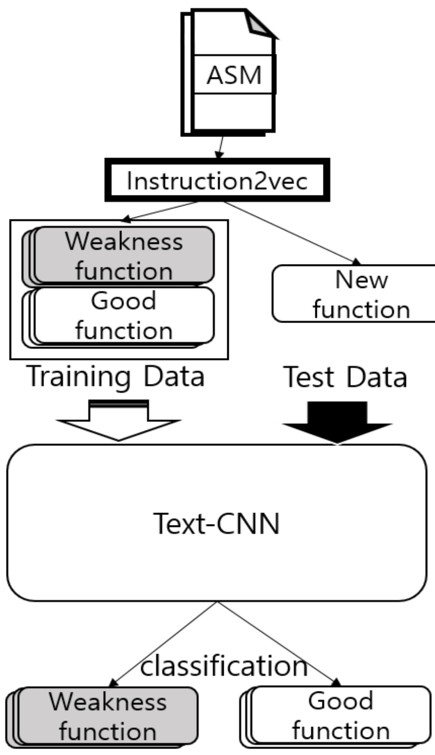

Fig. 1. Overall flow chart

3.1 Comparing *Word2vec* to *Instruction2vec*

This section explains existing studies about *Word2vec*, and introduces *Instruction2vec* newly proposed by us. *Word2vec* is a two-layer neural net that processes text. Its input is a text corpus and its output is a set of vectors. In fact, *Word2vec* contains two distinct models; a continuous bag of words and a skip-gram. The model learns to map each discrete word id, 0 through the number of words in the vocabulary, into a low-dimensional continuous vector space from their distributional properties observed in a text corpus. *Instruction2vec* includes the process of *Word2vec*. This is because the process of *Instruction2vec* utilizes the *Word2vec* results for all words in the assembly code. *Instruction2vec* generates a lookup table by replacing the assembly code which consists of four parts: opcode, register, pointer value, and library function, with vectors. This lookup table is used to match an instruction to a vector. Because instructions have a fixed length, a new vector is created when each vector is connected into a single entity. As a result, the new vector, which has a fixed length, is the value which efficiently expresses the instruction.

3.2 Principal of *Instruction2vec*

The purpose of *Instruction2vec* is to vectorize the instructions of the assembly code. There are two important characteristics of assembly code. The first characteristic is that its instructions have a fixed syntax. Most instructions use one opcode and two operands which also have a fixed length. Therefore, it is limited in length. The second characteristic of the assembly language is that it has significantly less number of words. The words are opcode, register, and library functions. NLP uses a high-dimensional vector to represent a large number of word vectors, but the assembly code does not need a high-dimensional vector because the number of words is small. That is why we propose *Instruction2vec* to efficiently extract features of assembly code. Most instructions have the form shown in **Fig. 2**.

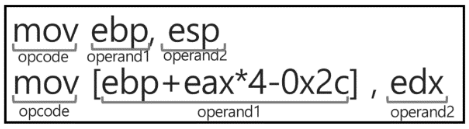

Fig. 2. Form of instruction

Instructions are formulated differently from natural language in assembly language grammar. It has a fixed length and only certain values can be located at certain positions. There is one opcode, and up to four operands, in general. Based on these characteristics, *Instruction2vec* vectorizes assembly code efficiently. The *Instruction2vec* process we designed is shown in **Fig. 3**. The opcode is in the first column. This means that there can be only one command, and that it is expressed in one column. Operand1 occupies the second through fifth columns. It is expressed in 4 spaces, and it can contain the name of a register, memory address, or a constant integer. Operand2 is the sixth through the ninth columns for the same reasons as Operand1.

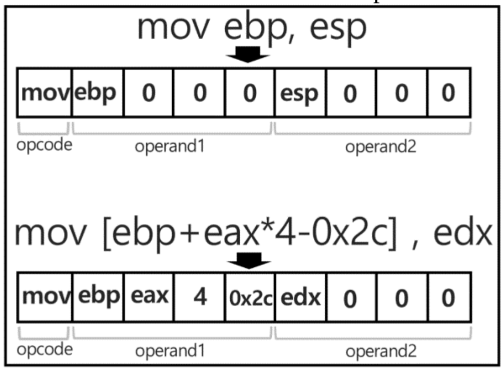

Fig. 3. Process of the *Instruction2vec*

We create a new vector, accumulating the vectors corresponding to the opcode, register, and address. For example, if each element of the instruction is filled with a vector as above, a new vector of 1*9 (*Word2vec* size is assumed to be 1) is created. If 1*9 represents a single line of instructions, then a dimension of N*9 is needed to represent the N lines of instructions. Hence, a function that has N lines can be expressed as a dimension of N*9, and the dimension is used as an input data for Text-CNN.

3.3 Process of the classification model

Here, we present the process of learning and classifying the created dataset via Text-CNN. Using *Instruction2vec*, we extract the features of assembly code and create a dataset. Training data in the dataset are labelled as either 'weakness function' or 'good function', and used as input to Text-CNN. The model repeatedly conducts the convolutional and subsampling operations and optimizes the weight values in each layer. In short, we train Text-CNN with a layer of convolution on top of instruction vectors. The model uses multiple filters to obtain multiple features; generally, one feature is extracted from one filter. These features help Text-CNN distinguish between 'weakness function' and 'good function'.

4. Experiment

This section explains about the experiment procedures and results. We created two datasets by using *Word2vec* and *Instruction2vec* for comparative experiment. We use 80% of the dataset for training, and holdout 20% for testing. Each dataset is used as input to CNN and Text-CNN for testing. The experiment results show that Text-CNN trained with *Instruction2vec* dataset has the highest accuracy.

4.1 Dataset

To test our model, we create a dataset by using *Juliet* test suite from National Institute of Standards and Technology. The *Juliet* test suite has example codes which contain software weaknesses per Common Weakness Enumeration (CWE). The examples are labelled as 'good case' or 'bad case', and hence are suitable for training. We choose testcase CWE-121, which contains stack-based buffer overflow vulnerability, among various test cases of the *Juliet* test suite. We use total 3474 functions from the testcase; Functions that validate the code length is excluded from our test. Functions compiled to 32 bits are disassembled to convert from elf file format to assembly code. The assembly code extracted from each function is preprocessed, for training, with *Instruction2vec* and *Word2vec* respectively to make two datasets.

4.2 Environment

We use an instance with NVIDIA Tesla K80. The operating system is Ubuntu 16.04.3 with four vCPUs and 16GB memory. We construct a neural network model using Tensorflow. CNN consists of two hidden layers. The first layer consists of thirty-two 3 by 3 filters, and the second layer consists of sixty-four 3 by 3 filters. Text-CNN uses 9 types of filters, each 128 in count, to maximize performance. The filters are of size 2, 4, 6, 8, 10, 12, 14, and 16, respectively.

4.3 Result

The result of our experiments shows that the Text-CNN model with *Instruction2vec* has the highest accuracy of 96.1%. As shown in **Table 1**, CNN and Text-CNN show significant differences in results from our experiments. Furthermore, in the case of Text-CNN, we can see the difference in accuracy between *Instruction2vec*s and *Word2vec*.

Table 1. Result of experiment

Dataset	Neural network	accuracy
Instruction2vec	CNN	87.6%
Instruction2vec	Text-CNN	**96.1%**
Word2vec	CNN	87.9%
Word2vec	Text-CNN	94.2%

5. Conclusion

In this paper, we propose a method of learning assembly code using deep learning to find software weaknesses. Especially, in case of *Instruction2vec*, the experiment results showed higher accuracy compared to that of *Word2vec*. Hence, we can conclude that *Instruction2vec* express assembly code into vectors more efficiently than *Word2vec*. Experiments on CNN and Text-CNN also demonstrate that Text-CNN shows better results in solving the classification problem for text data. In the future, we will expand our research to find vulnerabilities not only in functions, but in the entire program to include control-flow and data-flow of the program.

References

[1] KIM, Yoon, "Convolutional neural networks for sentence classification." *arXiv preprint arXiv:1408.5882*, 2014.

[2] Yamaguchi, Fabian, Felix Lindner, and Konrad Rieck, "Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning.", *Proceedings of the 5th USENIX conference on Offensive technologies*, USENIX Association, pp. 13-13, 2011.

[3] Grieco, Gustavo, et al, "Toward large-scale vulnerability discovery using Machine Learning," *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, ACM, pp 85-96, 2016.

[4] Goldberg, Yoav, and Omer Levy, "*Word2vec* Explained: deriving Mikolov et al.'s negative-sampling word-embedding method.", *arXiv preprint arXiv:1402.3722,* 2014.

[5] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton, "Imagenet classification with deep convolutional neural networks.", *Advances in neural information processing systems*, pp. 1097-1105, 2012.

[6] *Juliet* Test Sutie, https://samate.nist.gov/SRD/testsuite.php

[7] Kolosnjaji, Bojan, et al. "Empowering convolutional networks for malware classification and analysis." *Neural Networks (IJCNN), 2017 International Joint Conference on*. IEEE, 2017.

[8] White, Martin, et al, "Deep learning code fragments for code clone detection.", *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ACM, pp. 87-98, 2016.

[9] Gibert Llauradó, Daniel, "*Convolutional neural networks for malware classification*.", MS thesis, Universitat Politècnica de Catalunya, 2016.

[10] Cha, Sang Kil, et al, "Unleashing mayhem on binary code", *Security and Privacy* (SP), 2012 IEEE Symposium on. IEEE, 2012.

[11] Jo, Hwiyeol, et al, "Large-Scale Text Classification with Deep Neural Networks.", *KIISE Transactions on Computing Practices*, Vol. 23, No. 5, pp. 322-327, 2017