Embedding Source Code to Image for Defect Analysis

Xiaomeng Wang[a, *], Wei Xin[b], and Jiajie Wang[c]

China Information Technology Security Evaluation Center, Beijing 100085, China

[a]xiao_meng_wang@163.com, [b]xinwei@pku.edu.cn, [c]wangjj@itsec.gov.cn

*Corresponding author

Keywords: static code analysis, embedded images, convolutional neural networks, function level, cross file, cross projects

Abstract: Source code analysis can predict defective code regions, help developers to fix bugs and prioritize test overhead. Learning based methods fail to achieve promising performance on cross-file and cross-project tasks. Meanwhile, existing code review software works seriously depend on specific compiler or virtual compiler. If source code cannot be compiled the tools do not work. We first propose to extract function blocks from abstract syntax trees, embed them to code image and construct convolutional neural networks to learn embedded images deep, which produce some categorical models for source code defect classification. We have evaluated the proposed methodology on some experiments to analysis memory related defects in snippets function by function. Experimental results showed without compiling source code it is analyze some and find historical code defect such as CVE-2018-0732, CVE-2018-0737 and CVE-2018-0739.

1. Introduction

Source code defects or flaws probably lead to software vulnerabilities can potentially result in huge damages to businesses and people live, especially with the explosive growth of software. Once software attacked and exploited, it become threaten to society security including business, government and citizen property. Like the exposed Openssl Heartbleed in 2014, which has affected billions of consumers in university, enterprise and governments. Various defect prediction techniques have been proposed along the software life cycle to reduce the damages from software vulnerability [1].

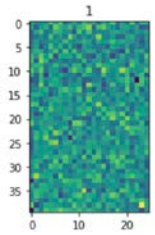
Static source code analysis (SCA) provides to prevent vulnerability eruption in primitive stage of software development. Classical SCA techniques commonly formulate rules from software data, like source code, introduction documents and released programs, to distinguish whether given snippets contain flaws or not. Recently attempts on SCA mainly fall into two categories: one is to explore new representation of code defects or make some combination of existing code features pursuing more effective expression like Halstead features based on operator and operand counts [2], McCabe features based on dependencies [3], CK features based on function and in- heritance counts, etc. [4], MOOD features based on polymorphism factor, coupling factor, etc. [5]. The other exploited learning based algorithms to acquire code patterns for defect classification or detection, including Support Vector Machine (SVM) [6], Naive Bayes (NB) [7], Decision Tree (DT) [8], convolutional Neural Networks (CNNs) [9], and recurrent Neural Networks (RNNs) [10].

Unfortunately, existing features cannot deal with code snippets analysis of different functions. Source files with different functions curved same values with some traditional features. Semantic features contained in source code is capable of distinguish aforementioned functions, learning based algorithms have been employed to learn semantic representation from program abstract syntax tree (AST) automatically to improve defect detection [11]. However, these methods do not show an intuitive expression to understand. Additionally, a realistic factor of compiling dependency result in complex pre-review work and waste a lot time. For instance, famous code review tools like Fortify and Coverity, before scanning program source code must be compiled or configured, otherwise the tools cannot work. The configuration and compiling take more time than code review sometime,

meanwhile these tools do not support code snippets audit.

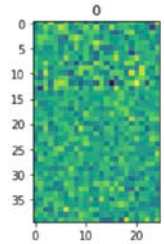
In summary, these factors lead to high time consuming and limited adaptation scenarios in code scanning. During research on source code analysis we found it surprise when embedding code to images, it shows indistinguishable difference to the naked eye between vulnerable code and non-vulnerable code showed in Figure 1. To explore accurate prediction models, we propose to embed source code to image and construct a deep learning model for defect code classification in function level without code compiling dependency.

(a1)Vulnerable code



(a2) embedded image of (a1)

(b1) Non-vulnerable code



(b2) embedded image of (b1)

Figure 1. An example of motivation

2. Embedding Code to Image

Image hold on sufficient information of both salient and latent features in the way of statistics. Vulnerable code represented obvious difference from normal code which is indistinguishable to the naked eye in embedded images. In this paper, we first propose to exploit image analysis in SCVD from three aspects: embed function to vector, transform vector to image and image analysis.

2.1 Embedding code to vector

Distributed representations of words, sentences, paragraphs, and documents played a key role in unlocking the potential of neural networks for NLP tasks. Methods for learning related distributed representations can produce low-dimensional vector representations for objects, referred to as embeddings, such that semantically similar objects are mapped to close vectors.

To embed code by vector representation, the objective is to make the average log probability maximum when giving a code snippet f1, f2… fn, n is a real number.

$$f_v = \frac{1}{T}\sum_{m=k}^{T-k} logp(f_m|f_{m-k},\dots,f_{m+k})$$

$$c^n = \{f_1, f_2, \dots, f_m\}$$

Put it as a classification problem, vulnerable code prediction can be done by a softmax classifier:

$$p(f_m|f_{m-k},\dots,f_{m+k}) = \frac{e^{y_{f_m}}}{\sum_i e^{y_i}}$$

y_i represents for the log probability without normalization for each code word i, calculated as:

$$y = b + Uh(f_{m-k},\dots,f_{t+k};W)$$

Where U and b are $softmax$ parameters. h is a average word vector extracted from matrix W, which is generated from word2vector framework.

Word vectors usually contribute to predicting next word in sentence. Code snippet embedding task stands on the shoulders of giants is a kind of paragraph vector model implementation for classification rather than predicting next word in classical way. In our work, a code snippet is defined as a function block extracted from AST, transformed to a long code sequence split by space and tokenized. Then we embed code sequence to matrix D with a fix size of L*M, meanwhile, each code is encoded by a vector of length L. The most significant adjustment is that h is a combination parameter constructed from *W* and *D* otherwise single *W*.

2.2 Normalize vector to image

W and *D* show abstraction of code snippet which is difficult to understand. Fortunately, image provide more intuitive pattern to distinguish whether it is normal or buggy in statistics, such as color space. Figure 1 displayed an example of motivation. Vectors from code level and snippet level embedding preserve adequate semantic information. Images are also matrix or vectors in nature. The main difference for people is that W or D yields an one dimension vector, image falls to M*N dimension matrix by default and for color image with an extensive channel C in various color spaces.

With the foundation of semantic feature, we exploit image features to predict code defect. Suppose F_s means the combination of W and D; the corresponding snippet image I_s can be embedded as:

$$I_s = f(F_s)$$

f is a normalization function to transform W and D into color space with a 3 channels color image of M*N*C dimension. For instance, in RGB color space each pixel various from 0 to 255 with 3 channels. Therefore, we map elements in W and D in range 0 to 255 to show its color density. In simplest case, we regularize vectors to M*N dimension of 1 Channel (C=1). In the normalized image, each pixel is described as $I(x, y)$:

$$I(x, y) = 255 * (v_i - v_{min})/(v_{max} - v_{min})$$

Ultimately, the collection of $I(x, y)$ contribute to a code snippet image *I*.

3. Experiments and results

We carry on several experiments to evaluate the performance of the proposed image based deep learning method and make comparison with existing learning based models. We conduct experiments on a Linux server with 4*Titan Xp GPU, 2* E2683 v4 CPU, 256 GB RAM, 2T SSD and 4T SATA disk and Ubuntu desktop system 16.04.

3.1 Datasets

We construct training set, validating sets and test sets from the Software Assurance Reference Dataset (SARD), which provides users, researchers and software security assurance tool developers with a set of known security flaws. This will allow end users to evaluate tools and developers to test or evaluate their methods. These test cases are designed source code and binaries from all the phases of the software life cycle. Meanwhile, it includes "wild" (production), "synthetic" (written to test or generated), and "academic" (from students) test cases. This database will also contain real software application with known bugs and vulnerabilities, which makes contribution to annotating the extracted functions. Test sets and training sets are split by 1/4, which lead to 98004 functions for test and 392016 functions for training. During training state 5-fold cross validation was implemented.

3.2 Model Setting

The proposed method transform context to image for vulnerability discovering task. Finally, we also compare aforementioned model with our previous work, which utilize code property graph to detect defect code. The results of efficiency evaluation. All the functions are extracted in SARD, each function is labeled by ‘0’ of normal code or ‘1’ of vulnerable code. Then we embed functions to images of 40*25 in pixel as DLM input, batch size is set of 128 and epoch number is 100,

additionally Adam is selected as the optimizer and in the last activation layer sigmoid is employed.

Table.1. Results of efficiency evaluation (%)

	Loss	Accuracy	Recall	Fmeasure	FP	FN	TP
training	2.54	99.06	98.97	99.01	0.85	1.03	98.97
validation	9.721	84.71	85.91	84.25	16.42	14.09	85.91

4. Results and conclusion

We get corresponding evaluation results showed in TABLE 1 with a custom CNN model with 5 convolution layers and for each following with a max pooling operation for SARD datasets. Experimental showed the proposed method is effective for code defect analysis by embedding them to images. Even more surprising is that we performed defect code detection on Openssl 1.0.2n and found some confirmed vulnerable code such as CVE-2018-0732, CVE-2018-0737 and CVE-2018-0739.

Through exhaustive illustration, the insight of embedding source code to image for defect analysis using CNN works well in code snippet which survive without compiling source code. Unfortunately, data scale and annotation constrained the effectiveness of the method. In future, we will focus on expanding source code database scale and exploring unsupervised method to further optimize the image based source code analysis.

Acknowledgments

This paper was supported by National Natural Science Foundation of China (NSFC) U1836209, U1636115 and U1836113.

References

[1] Ramanathan Ramu, Ganesha Upadhyaya, Hoan Anh Nguyen, Hridesh Rajan. Hybrid traversal: efficient source code analysis at scale. ICSE (Companion Volume) 2018: 412-413

[2] M. H. Halstead. Elements of Software Science (Operating and programming systems series). Elsevier Science Inc., 1977.

[3] T. J. McCabe. A complexity measure. TSE'76, (4): 308–320.

[4] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. TSE'94, 20 (6): 476–493.

[5] R. Harrison, S. J. Counsell, and R. V. Nithi. An evaluation of the mood set of object-oriented software metrics. TSE'98, 24 (6): 491–496.

[6] Vera Barstad, Morten Goodwin, Terje Gjøsæter. Predicting Source Code Quality with Static Analysis and Machine Learning. NIK 2014

[7] Siegfried Rasthofer, Steven Arzt, Eric Bodden. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. NDSS 2014

[8] Qingkun Meng, Chao Feng, Bin Zhang, and Chaojing Tang. "Assisting in Auditing of Buffer Overflow Vulnerabilities via Machine Learning" Mathematical Problems in Engineering, vol. 2017, Article ID 5452396, 13 pages, 2017.

[9] Lili Mou, Ge Li, Lu Zhang, Tao Wang, Zhi Jin.Convolutional Neural Networks over Tree Structures for Programming Language Processing. AAAI 2016: 1287-1293

[10] Song Wang, Taiyue Liu, Lin Tan. Automatically learning semantic features for defect prediction. ICSE 2016: 297-308

[11] Uri Alon, Meital Zilberstein, Omer Levy, Eran Yahav. code2vec: Learning Distributed Representations of Code. CoRR abs/1803.09473 (2018)