Vulnerability Detection with Deep Learning

Abstract

As the number of softwares being produced and the size of each software grow, vulnerabilities in softwares are also becoming more prevalent. In order to detect these vulnerabilities, efficient detection techniques are required. How to detect vulnerabilities has been worked on for a long time and various approaches have been introduced, for instance, static or dynamic program analysis. Although they had achieved their own success, they are still practically infeasible to be applied to complex modern softwares. Various requirements such as modest execution time, accuracy and granularity should be met to efficiently detect vulnerabilities. To achieve such objectives, I propose vulnerability detection method combining comprehensive representation and deep learning. I use code property graph as such representation, and compare various deep learning models to reach my objectives.

1. Introduction

Modern softwares are as complex as ever. As they rapidly grow in quantity, flaws in such systems are becoming more prevalent. Since a single defect can compromise the entire system, detecting these flaws is an important task. While most of these system flaws are just minor bugs, some can be controlled by malicious attackers and permit unauthorized actions. Thus, it is important to find such critical flaws, which are called *vulnerabilities*. In linux kernel, hundreds of such vulnerabilities are found every year [1]. Due to the increasing number of such vulnerabilities followed by the increase of system flaws, we need an automated way of detecting software vulnerabilities.

Various methods of detecting vulnerabilities have been introduced such as static and dynamic analysis, and other methods such as metric-based detection. Static analysis includes symbolic execution which provides exact attack vector with high accuracy. However, execution time of symbolic execution dramatically increases when the target software is complex, making it impractical to be applied to modern softwares. Dynamic analysis such as fuzzing can find vulnerabilities in reasonable time. Nevertheless, some softwares are difficult to be analysed dynamically, and dynamic analysis cannot be fully automated since they require manual specifications. In order to resolve such limitations of existing methods, I propose vulnerability detection model combining comprehensive representation and machine learning. I use code property graph, a graph data structure presented to model vulnerability in software code, as input representation to my learning model. This intermediate representation shows control flow or data dependency more explicitly than raw text of source code. I also introduce various machine learning models to process such representation.

I present the following contributions in this work:

- I suggest how the software code should be transformed to train vulnerability detection model. I use code property graph as such representation, and show that code property graph provides necessary information to detect vulnerabilities in software with high granularity and accuracy.
- I investigated various neural network models to process the graph data. I implemented the model which extends convolutional neural network to process arbitrary graphs.
- I generated dataset labeled with the type of vulnerability from open source repositories. Compared to existing datasets used in classical software defect prediction, my dataset is labeled in function-level,

thus enabling learning model to detect vulnerability with higher granularity.

The rest of the paper is organized as follows: Section 2 describes background knowledge to understand vulnerability detection and neural network. Section 3 describes my problem domain and requirements to my objectives. Section 4 introduces previous works on vulnerability detection and their limitations I focus on. Section 5 suggests my approach on designing new detection model including code property graph and neural network for graph data. Section 6 introduces my methods to generate an appropriate software code dataset for my learning model. Section 7 describes detailed implementation of my model followed by related works in Section 8. Finally, I conclude my work with some future works in Section 9.

2. Background

In this section, we provide background knowledge about vulnerability and neural network which is also known as deep learning.

2.1 Neural Network.

Neural network is a model used in machine learning. Nodes in neural network are called neurons. Connections between neurons are used to propagate values to each other, with value of each edge works as a weight of propagation. Adjusting weights of each connection by propagating forward and backward, the neural network 'learns' how to evaluate desired output features from features contained in input data. Neural network can be structured as multiple layers and operates propagation between adjacent layers. Model with more layers can represent more complex attributes, which are called deep learning models.

2.2 Convolutional Neural Network.

Convolutional neural network (CNN) is a deep learning model which is widely used in image and video recognition and natural language processing. Typically, the CNN takes 2-dimensional data, usually image, as input and feed forward them in network with convolution operations. A convolution operation takes a small region in input and reduce the region into single value. The convolution layer works as a feature extractor, reducing the complexity of input by preserving desired high level features only.

3. Problem Definition

Vulnerability requires three elements: a flaw in system, accessibility to the flaw, and capability to exploit the flaw. In this paper, I focus on inherent vulnerabilities in software such as memory corruption.

I aim to detect software vulnerability with sufficient accuracy and granularity in moderate execution time. Although existing methods have achieved efficiency to some extent, each has their own limitations which will be described in the following section.

4. Previous Research

How to detect vulnerabilities in software has been studied long and diverse methods have been introduced such as static or dynamic analysis, formal verification, and a variant of bug detection methods. In this section I will introduce three types of typical previous research, static analysis, dynamic analysis, and metric-based vulnerability detection, and motivate the need for a new means of vulnerability detection by showing their limitations.

4.1 Static Analysis

Static analysis is a way of analysing software without running the software itself. This includes symbolic execution such as KLEE [4], which determines what input leads to certain part of the program by using symbolic values for variables. It can detect vulnerability with specific attack vector and clearly show how it exploits the program. However symbolic execution has limitation known as path explosion which makes using symbolic execution on large softwares practically infeasible.

Other methods of static analysis include formal verification, which proves program specification based on mathematical design. Although being applied in recent softwares [2], formal verification has difficulty of designing program specifications into mathematical form and proving them, both of which cannot be done in automated way.

4.2 Dynamic Analysis

In contrast to static analysis, dynamic analysis executes the software for analysis and observes its behaviour. Dynamic analysis can be used in memory error detection [14], software testing associated with code coverage [9], or analysing program behaviour [6, 16]. However dynamic analysis requires manual specification of

```
int f(bool x)
{
  if (x) {
    return 1;
  }
  else {
    return 2;
  }
}
```

Figure 1: Exemplary code sample.

detection, which makes automating dynamic analysis difficult. Moreover, they cannot consider every possible attack vectors.

4.3 Metric-based Analysis

Metric-based detection method uses code features such as lines of code(LoC) or function call dependency to detect vulnerabilities. Neuhaus et al. [15] designed predictor detecting vulnerable components based on function calls. Despite metric-based detection can be executed fast, it lacks detection granularity and accuracy.

Other recent research includes detecting buffer overruns from source code using neural memory networks [5], however it cannot be extended to general types of vulnerabilities.

5. Approach

To address the limitations identified in the previous section, we need representation more comprehensive than those used in existing vulnerability detection methods. Execution time of detection should also be scalable. Deep learning can achieve both effectiveness and efficiency of detection. Shin et al. found that applying deep learning for software analysis has advantages in many perspectives [19]. To use deep learning, how to represent input data and learning model should be decided.

The way of representing dataset, i.e. softwares, should meet the following criteria:

Representation Granularity. Each element of representation should include information of software in order to achieve function-level granularity. Representation should be able to determine in which function the vulnerability resides, which cannot be accomplished with simple code metrics such as LoC. Therefore, information of software should be distributed to detailed component of program structure to help model determine the exact location of vulnerability. Both the raw form such as binary [11, 19] and preprocessed form such as abstract syntax trees [20] have been used as representation of software for deep learning models.

Program Context. Since same code can be either safe or vulnerable depending on the context, detailed information of control flow and dependence of program are also required. This difference is, although inherent, not distinguished explicitly by program structures only. Hence, such information should also be shown in representation.

5.1 Representation

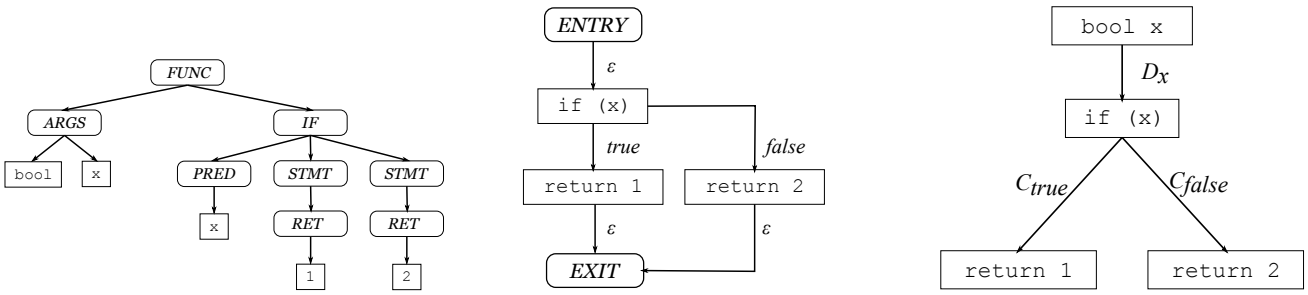
I use code property graph to satisfy requirements described above. Code property graph was introduced to effectively mine large amounts of source code for vulnerabilities [21]. Code property graph combines abstract syntax tree, control flow graph and program dependency graph. Thus, syntactic structure, control flow and dependency between each part of software code are included, successfully meeting requirements mentioned above.

Abstract Syntax Tree (AST) Abstract syntax trees represent structure of source code in an abstract way. Figure a shows example AST for the code sample Figure 1. Their nodes do not necessarily correspond to exact syntax token of the program but can represent larger semantic unit such as condition expression or variable declaration.

Control Flow Graph (CFG) Control flow graph describes conditions and order of each execution path of program. Figure b shows example CFG for the code sample given above. While control flow graph provides information of condition and context, data flow is still required to determine the exact attack vector.

Program Dependency Graph (PDG) Program dependency graph consists of two types of edges: data dependency edges and control dependency edges. Figure c shows example PDG for the code sample given above, data and control dependency edges are labeled as D and C respectively. Data dependency edges connect variable values and statements affected by them. Control dependency edges are different from control flow graph since the former do not contain execution order but show dependencies more clearly.

The combination of three graphs forms code property graph shown in Figure 3.



(a) Abstract Syntax Tree **(b)** Control Flow Graph **(c)** Program Dependency Graph

Figure 2: Representations of code for the example in Figure 1.

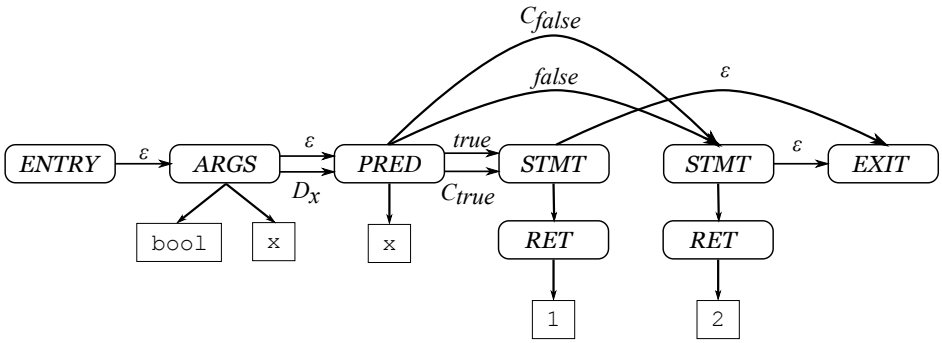


Figure 3: Code Property Graph

5.2 Learning Model for Graph Data

Since we use code property graph as program representation, we need neural network model which takes graph data as input. Code property graph is a directed multigraph with edges containing multiple, isolated attributes. We consider typical learning models for graph data:

Support Vector Machine with Graph Kernel. Support vector machine (SVM) is a type of machine learning that can classify input data. SVM calculates decision surface using cross products of inputs. Although SVM is not deep learning method, it can be used as baseline. Kernel in SVM is a function that adjusts input data or their cross products. Kernel is used to help SVM to classify data more efficiently, by mapping different inputs more distinctly. Graph kernel, a kernel function for graph input, has been mainly applied in bioinformatics and cheminformatics, for instance, predicting protein function by its structure. Classical graph kernels are based on walks, paths, subgraphs or subtrees.

Weisfeiler-Lehman (WL) graph kernel [18] is state-of-the-art among graph kernels. WL graph kernel generates feature vector from subtree pattern in every iteration. In algorithm, neighborhood information is compressed to label of each node. Feature vector are then created in each iteration, consists of numbers of each label appeared during the algorithm. The similarity between graphs are calculated as the inner product of their feature vectors.

Graph Neural Network. Graph neural network (GNN) is a learning model which directly uses input graph layout as model [8]. GNN extends recursive neural networks, using nodes of input graph as state representations. For each iteration, state values are updated by being propagating by other connected states. GNN eventually generates some desired attribute of the input graph. Li et al. introduced Gated graph sequence neural network (GGS-NN) based on GNN. GGS-NN is an extended model of GNN which can find the output sequence of graph producing such attribute [13].

Convolutional Neural Network for Graph. Niepert et al. presented an approach to process general graphs with CNN by considering input of classical CNN as lattice graph [17]. From such perspective, receptive field of classical CNN is a neighborhood subgraph from certain node with fixed width. Thus for an arbitrary input graph, receptive field can be

generated and consequently passed as input to classical CNN.

6. Dataset

Dataset is an important factor of learning the model. In order to detect vulnerabilities with function-level granularity and to classify them with type, we need each function labeled either secure or not, and with type of vulnerability if insecure.

Despite the number of available source codes, labeling them is challenging. Tera-PROMISE [3] is a research dataset repository for software engineering. Although it has been widely used in area of metric-based bug detection, its datasets are labeled with code metrics which cannot be used to our model. Neuhaus et al. used vulnerabilities database from Mozilla project to predict vulnerable component based on function calls, but granularity still remains at module level [15].

Labeling certain function code as whether secure or not can also be a challenge. If there is an exact attack vector, it can be clearly shown that the code has vulnerability, while strictly proving it doesn't can only be done by formal verification. I assumed that code after patching vulnerabilities does not have vulnerabilities anymore.

Due to limitations of existing datasets described above, I collected new source code dataset from Github. I collected merged pull requests of which title containing vulnerability type (e.g., buffer overflow). Then I labeled merged commit as secure and base commit as insecure.

7. Implementation

We implemented vulnerability detection model as convolutional neural network. Dataset of source codes are first converted by Joern [21]. Joern generates code property graphs from each C/C++ source code repository. Since feeding the whole repository is inefficient, I parsed them to contain relevant code sections which are diffs between base and merged commit. The following stages of learning are implemented as same as PATCHY-SAN [17].

8. Related Work

Malware Detection with Machine Learning. Droidsec [22] detects malware based on features extracted from static and dynamic analysis of Android application. Kosmidis [11] transformed program binary into digital image, and detected malware pattern from image using machine learning.

Software Defect Prediction. Software defect prediction [10, 12] is one of classic bug detection methods. Gao et al. [7] considered various feature subset selection methods to efficiently detect bugs. Wang et al. [20] extracted semantic code features using deep learning.

9. Conclusion

In this paper, I suggest vulnerability detection model that using deep learning to detect vulnerabilities efficiently and effectively. Detection model combines code property graph as a comprehensive representation and convolutional neural network for graph data to process the code property graph. Evaluation of this model is left for future work. Furthermore, other detection models with different learning models must be compared.

References

[1] Linux linux kernel: Cve security vulnerabilities, versions and detailed reports. http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33.

[2] Rustbelt. http://plv.mpi-sws.org/rustbelt/.

[3] The promise repository of empirical software engineering data, 2015.

[4] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[5] M.-j. Choi, S. Jeong, H. Oh, and J. Choo. End-to-end prediction of buffer overruns from raw source code via neural memory networks. *arXiv preprint arXiv:1703.02458*, 2017.

[6] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[7] K. Gao, T. M. Khoshgoftaar, H. Wang, and N. Seliya. Choosing software metrics for defect prediction: an investigation on feature selection techniques. *Software: Practice and Experience*, 41(5):579–606, 2011.

[8] M. Gori, G. Monfardini, and F. Scarselli. A new model for learning in graph domains. In *Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on*, volume 2, pages 729–734. IEEE, 2005.

[9] C.-Y. Huang, C.-H. Chiu, C.-H. Lin, and H.-W. Tzeng. Code coverage measurement for android dynamic analysis tools. In *Mobile Services (MS), 2015 IEEE International Conference on*, pages 209–216. IEEE, 2015.

[10] T. M. Khoshgoftaar, L. A. Bullard, and K. Gao. Attribute selection using rough sets in software quality classification. *International Journal of Reliability, Quality and Safety Engineering*, 16(01):73–89, 2009.

[11] K. Kosmidis. Machine learning and images for malware detection and classification. 2017.

[12] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4):485–496, 2008.

[13] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[14] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[15] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 529–540. ACM, 2007.

[16] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[17] M. Niepert, M. Ahmed, and K. Kutzkov. Learning convolutional neural networks for graphs. In *Proceedings of the 33rd annual international conference on machine learning. ACM*, 2016.

[18] N. Shervashidze, P. Schweitzer, E. J. v. Leeuwen, K. Mehlhorn, and K. M. Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(Sep):2539–2561, 2011.

[19] E. C. R. Shin, D. Song, and R. Moazzezi. Recognizing functions in binaries with neural networks. In *USENIX Security*, pages 611–626, 2015.

[20] S. Wang, T. Liu, and L. Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*, pages 297–308. ACM, 2016.

[21] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 590–604. IEEE, 2014.

[22] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue. Droid-sec: deep learning in android malware detection. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 371–372. ACM, 2014.