An Extended Benchmark System of Word Embedding Methods for Vulnerability Detection

Hai Ngoc Nguyen
Graduate School of Information Science and Engineering, Ritsumeikan University
Shiga, Japan
hai@cysec.cs.ritsumei.ac.jp

Hoang Viet Nguyen
Graduate School of Information Science and Engineering, Ritsumeikan University
Shiga, Japan
hoang@cysec.cs.ritsumei.ac.jp

Tetsutaro Uehara
College of Information Science and Engineering, Ritsumeikan University
Shiga, Japan
uehara@cysec.cs.ritsumei.ac.jp

ABSTRACT

Security researchers have used Natural Language Processing (NLP) and Deep Learning techniques for programming code analysis tasks such as automated bug detection and vulnerability prediction or classification. These studies mainly generate the input vectors for the deep learning models based on the NLP embedding methods. Nevertheless, while there are many existing embedding methods, the structures of neural networks are diverse and usually heuristic. This makes it difficult to select effective combinations of neural models and the embedding techniques for training the code vulnerability detectors. To address this challenge, we extended a benchmark system to analyze the compatibility of four popular word embedding techniques with four different neural networks, including the standard Bidirectional Long Short-Term Memory (Bi-LSTM), the Bi-LSTM applied attention mechanism, the Convolutional Neural Network (CNN), and the classic Deep Neural Network (DNN). We trained and tested the models by using two types of vulnerable function datasets written in C code. Our results revealed that the Bi-LSTM model combined with the FastText embedding technique showed the most efficient detection rate on a real-world but not on an artificially constructed dataset. Further comparisons with the other combinations are also discussed in detail in our result.

CCS CONCEPTS

• **Computing methodologies** → **Natural language processing**; **Information extraction**; **Neural networks**; • **Security and privacy** → **Software security engineering**.

KEYWORDS

Deep Learning, Word Embedding, Vulnerability Detection, LSTM, CNN

ACM Reference Format:
Hai Ngoc Nguyen, Hoang Viet Nguyen, and Tetsutaro Uehara. 2020. An Extended Benchmark System of Word Embedding Methods for Vulnerability Detection. In *The 4th International Conference on Future Networks and Distributed Systems (ICFNDS) (ICFNDS '20), November 26–27, 2020, St.Petersburg, Russian Federation.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3440749.3442661

ICFNDS '20, November 26–27, 2020, St.Petersburg, Russian Federation

ACM ISBN 978-1-4503-8886-3/20/11…$15.00
https://doi.org/10.1145/3440749.3442661

1 INTRODUCTION

Software vulnerabilities could significantly damage the services and activities of any organization when exploited by cyber-attacks. To improve the software quality, experts and researchers have extensively studied both dynamic and static methods for analyzing software code. Dynamic analyzers examine the program control flow and are usually developed from the rule-based solutions. This approach is limited to the number of known vulnerabilities, and updating rules requires expert knowledge. Static code analyzers, on the other hand, locate the vulnerabilities without executing a program source code. This direction has recently attracted more attention since software source code was proven to share many identical characteristics with natural language texts [2]. Particularly, the use of NLP applications in automatically detecting vulnerabilities in code has been investigated. As deep learning gain greater success in numerous fields, such as in NLP, research shows the further potential of deep learning application in static code analysis [25]. Any system built based on machine learning or deep learning would require a specified embedding technique to generate model inputs as vector representations. Notwithstanding, there are many existing embedding techniques in the NLP field such as Word2Vec [17] and GloVe [21]. Since the number of embedding methods is increasing, selecting a compatible method for specific neural models can be challenging.

Many deep learning models such as Text-CNN [10] and Long Short-Term Memory (LSTM) [27] were employed to learn the vulnerable patterns from distinguished program code representations [6, 12, 13]. Most of these studies used Word2Vec to produce code vector representations, but other embedding methods like GloVe or FastText [4] have yet to be evaluated on these models. Different kinds of knowledge representations, such as linguistic contexts of identifiers and their order sequences, can be extracted by changing either the embedding technique or the model structure. Therefore, selecting the most suitable embedding method can be a critical task since it can affect the vulnerability detection performance of the classifiers. In this work, we extended the open-source benchmark API [14] by evaluating three additional word embedding techniques. We also used the two types of C code datasets which were originally provided as the baselines for comparisons in the benchmark. We set up the experiments to explore the effectiveness of the word embedding techniques for building vulnerability detectors at the

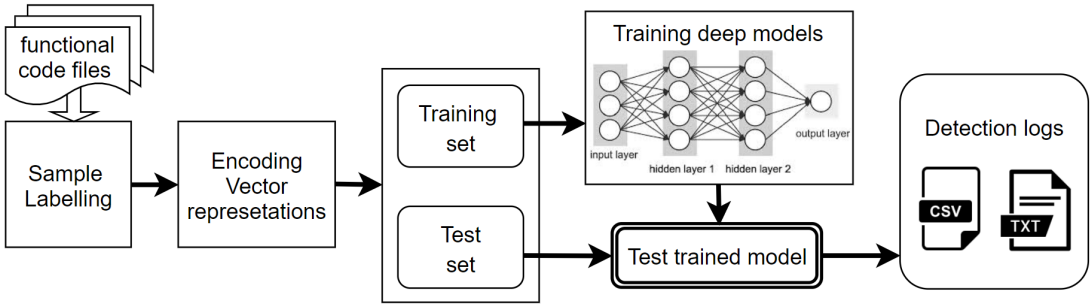

Figure 1: Approach Overview

function level. We employed the API to build a system to train and test the vulnerability detectors in a supervised manner of deep learning.

The main contributions of this paper are summarized as follows:

- We extend a benchmark system by evaluating three additional word embedding methods to generate input vector representations from the C program functions.
- We train and test the Bi-LSTM with the Hierarchical Attention Network (Bi-LSTM HAN), Bi-LSTM, Text-CNN, and DNN models on two different datasets.
- We carry out an overall performance evaluation of all trained models. Specifically, each model is analyzed with distinguished input representations to explore the effective combinations of the embedding techniques and the models.

The rest of this paper is arranged as follows: Section 2 presents the related studies where the word embedding techniques and the studied deep learning models were applied. Section 3 describes the detailed design of our system. In section 4, we explain the experiments and performance metrics. Section 5 provides the results and its comparative analysis. We conclude our work and discuss future directions in Section 6.

2 RELATED WORK

Motivated by the success of NLP and neural language models, prior studies observed similarities in semantic and syntactic information between natural language to the programming language. They took advantages of the NLP applications for software vulnerability detection and defect prediction [18]. Later, researchers focused on experiments with more complex machine learning structures while employing distinctive embedding techniques for producing vector representations. For example, the Word2Vec embedding algorithm was implemented for encoding the code vectors extracted from the Abstract Syntax Trees (ASTs) contexts [22]. The neural network called feed-forward then used those vectors as inputs for training the JavaScript code vulnerability detectors. Similarly, Word2Vec was implemented for generating vector representations from C/C++ source code. These representations and their control flow graphs (CFGs) data were both utilized to train the vulnerability classifiers [8]. Another study used the GloVe model for encoding vector representations from the Abstracted Symbolic Traces of C programs [9]. Additionally, a vulnerability prediction system, which was developed based on ensemble machine learning algorithms, employed the FastText model for the encoding task [7]. Given these points, many word embedding techniques had been implemented for detecting vulnerabilities in software code. However, due to the variety of the dataset types and the machine learning model structures, comparisons between these techniques were not possible to accomplish.

The mentioned studies had shown promising results in the vulnerability detection task, but their trained models were evaluated on their self-constructed datasets such as ASTs or CFGs. The success of these methods were based on syntactic artificial datasets which may raise a question on whether these datasets are more useful than basic inputs such as word vectors. There are studies that have recently investigated the effectiveness of different representations for the neural networks to work on the software code classification problems. Specifically, a comparative analysis was conducted to assess how different deep learning models learn over distinctive input representations of Java code [23]. Furthermore, Lin (2019) proposed a benchmark framework and compared three models which are Text-CNN [10], DNN, and Bi-LSTM. The three models were trained solely on the Word2Vec embedding model, and further comparison of applying other embedding algorithms or different neural networks are yet to be explored.

3 APPROACH

3.1 Overview

We aimed to build a system to investigate the effectiveness of four different word embedding methods for training vulnerable code classifiers. Figure 1 shows the overview of our approach. In the initial stage, the system loaded the dataset files and transformed them into sequences of word tokens which originally are code identifiers, variables, etc. Each of these sequences stands for a semantic function representation. By processing the file names, the list of vulnerability labels was generated correspondingly. The next stage is the encoding phase. Here, the system executed the predefined word embedding algorithm to train and map the sequences of tokens into vector representations. These vectors will then be partitioned into training and test sets. Eighty percent of the vector representations are fed to a designated neural network for training the vulnerability

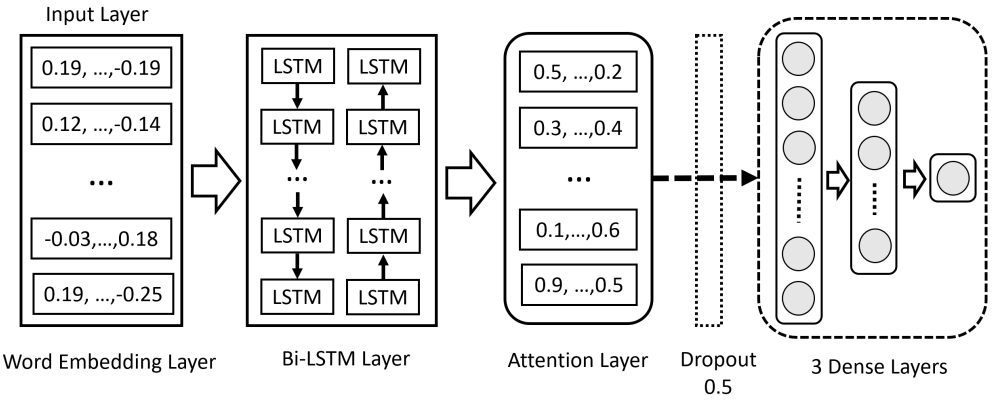

Figure 2: Bi-LSTM model with the Hierarchical Attention Network (Bi-LSTM HAN)

detector. When the training process was finished, the rest of the representations were tested by the trained detector. After the model was tested, the vulnerable probabilities of the test samples were generated correspondingly to a raw CSV file. Finally, our system called a script to process the CSV file data and sorted the list of test names to another table in the order of their vulnerability probability. The performance metrics calculation and detection logs were then automatically collected.

3.2 Encoding Vector Representations

In this phase, each of the sample files was eventually transformed into a vector representation. This representation is responsible for keeping both semantic and syntactic knowledge derived from the source code. In order to obtain the code representations, the embedding layer was initially defined in a configuration file. After the system finished generating the labels and extracting the sequences of code tokens from the dataset, the word embedding layer was then trained from all the extracted tokens. Next, the system loaded the trained embedding layer to transform the code sequences into the unified length vectors. Regarding the content of the two datasets, more than 90% of the samples were found to have the sequence length shorter than 1000. Therefore, we select the unified length of the code sequence to 1000 to balance the sparsity and the length of the dataset [15]. The functions which had the sequence length longer than 1000 are truncated to 1000. In contrast, the vector representations were appended zeros in case their sequence lengths were shorter than 1000. In our work, Word2Vec, GloVe, FastText, and GloVe pre-trained models (GloVePre) [21] were implemented. We configured the dimension setting (d= 100) as the default for all the embedding methods.

The Word2Vec model was originally implemented in the Lin API [14]. The model was provided by the GenSim [24] package. It provided two training algorithms namely Continuous Bag of Words (CBOW) and Skip-gram. Skip-gram aims to predict the context of a word, while CBOW learns to predict the word by its context. Hence, CBOW was selected to learn the syntactic code sequence knowledge but not its context. The embedding dimension size for the word vector is the same as d, and its *window* is 5. In addition, the GloVe was implemented by the glove-python package with the learning rate at 0.05. The *window* setting is 10, and the model was trained with four threads in 500 Epochs. The vector representations obtained from the GloVe model are quite different from Word2Vec since GloVe presented its representations by factorizing the logarithm of the corpus word co-occurrence matrix.

The GloVePre model was implemented to watch the baseline difference between converting words to vectors and code identifiers to vectors. The pre-trained layer was selected with the 100d pre-trained model $GloVe.6B.100d.txt$. Finally, the FastText model was constructed from the GenSim package. To train the FastText model, the number of threads and the *window* size were configured to 4 and 5 like in the Word2Vec model, while the rest of the parameters were set as default. In our work, FastText and GloVe were regarded as the main comparisons to the Word2Vec model in training the neural networks. Particularly, FastText can construct the vector for the word from its character n-grams even when the word is out of its vocabulary. Altogether, the code sequences were transformed into vector representations with the shape of (1000,10) by the selected embedding models.

3.3 Training Deep Models

The previous phase passes the meaningful code embedding vectors as inputs to train a predefined neural model. Particularly, the training and validation process utilized 80 percent of the embedding vectors. By learning from the extracted information from those vectors, the model could therefore differentiate the vulnerable and non-vulnerable code files. Our work concentrated on extending the evaluation of different code representations. We selected the Bi-LSTM, the Text-CNN, the DNN, and the Bi-LSTM HAN [6] models for learning the features extracted from the embedded code representations. The three preceding models were originally built in the API, while the Bi-LSTM model with Attention Mechanism was selected as an extra model for comparison.

The Bi-LSTM, Text-CNN, and DNN models in this work were designed in the same way as those in the original API [12]. Figure 2 illustrates the network architecture of the Bi-LSTM HAN model; it consists of six layers. The first layer, which takes the vector inputs

Table 1: The distribution of the vulnerable functions on two datasets.

Dataset	Training and validation set (unit: files)		Test set (unit: files)	
	The vulnerable number	Total number	The vulnerable number	Total number
The Nine- projects dataset	1155	48934	318	12234
The SARD dataset	31682	60000	3318	15000

Table 2: Performance Metrics.

Metric Name	Formula	Explanation
Precision at rank K	$P@K\% = \frac{TP@k\%}{TP@k\%+FP@k\%}$	The proportion of top-K functions that are actual vulnerable functions.
Recall at rank K	$R@K\% = \frac{TP@k\%}{TP@k\%+FN@k\%}$	The proportion of the relevant functions that are in the top-K.

from the embedding layers, is a bidirectional LSTM recurrent layer with 64 LSTM cells. The second layer is the attention layer which can provide key features for training more precisely the prediction model. The third layer is a dropout regulation layer with the dropout rate at 0.5. The last three layers are dense layers. The first dense layer contains 64 neurons, and the number of neurons is decreased by fifty percent in the next dense layer. The last dense layer has only one neuron and converges the output into a singly probability by using sigmoid activation function.

4 EXPERIMENTS

4.1 Experimental Setup

We set up the experiments to address the following research questions:

- Question 1: Would changing the embedding method improve the effectiveness of the vulnerability detector?
- Question 2: How does each combination of the embedding technique and the neural model perform when using different dataset?
- Question 3: Can changing the neural network structure affect detection performance?

In summary, we had trained and tested 16 classifiers for each dataset. In detail, we applied four embedding methods to train and test four different neural networks on two genres of datasets respectively. We configured the Stochastic Gradient Descent (SGD) as the optimizer for all the networks, followed by the default settings of Keras. The loss function was set to the binary cross-entropy algorithm. The deep learning models were implemented in Python (version 3.6.9) using Keras (version 2.2.4) with a TensorFlow backend (version 1.14.0) [1]. Word2Vec and FastText models were constructed by the GenSim pip library (version 3.4.0) while the GloVe model was used from the glove-python (version 1.0.1) package [11]. Our experiments were designed and carried out on an Ubuntu server (18.04 LTS) having 64GB RAM with an NVIDIA GeForce RTX 2080 SUPER 8GB GPU and an Intel(R) Core (TM) i7-9700K 3.60GHz CPU.

4.2 Datasets

In this work, we utilized two different vulnerability datasets for training the classifiers. Both datasets contain files written in the C program language where each file represents either a vulnerable or non-vulnerable function. The first dataset is the Nine-projects dataset which was originally proposed in the benchmark API [14, 19]. It was constructed from nine open-source projects with the vulnerability information extracted from the National Vulnerability Database [20] and the Common Vulnerabilities and Exposures [5] websites. The second dataset is obtained from the Software Assurance Reference Dataset [3, 26] project, which contains a large number of the artificially synthesized function files. The project covers the vulnerability test functions for C/C++ and Java. In this paper, we only used the C source code for our experiments. Following the studied experiments in the benchmark API, we randomly extracted 35000 vulnerable and 40000 non-vulnerable C function files from the SARD functions dataset provided by the same GitHub repository. The system loaded one dataset at a time. After being encoded to the labeled vectors, the dataset is distributed with the rate of 0.8 for the training and validation set, and 0.2 for the test set. The content of the datasets is described in Table 1. We use this data partition setting to train and test all the deep learning models.

4.3 Performance Metrics

Precision, Recall, and F1-score are generally used for evaluating the performance of deep learning models. Nevertheless, in many cases of vulnerability detection, the dataset imbalance between non-vulnerabilities and vulnerabilities showed that these metrics would undervalue the model detection performance [14]. Therefore,

Table 3: Distribution of precision and recall over top $k\%$ retrieved functions among the four models tested on the Nine-projects dataset: (1-4) DNN, (5-8) Text-CNN, (9-12) Bi-LSTM, (13-16) Bi-LSTM HAN. The highest values for each model are in bold.

Index	Model	Embedding	Dataset	Precision and Recall calculated for top k% retrieved functions							
				1%		10%		20%		50%	
				P_k%	R_k%	P_k%	R_k%	P_k%	R_k%	P_k%	R_k%
1	DNN	Word2Vec	9 Projects	34.4%	13.2%	**17.3%**	**66.7%**	**10.7%**	**82.1%**	**5.2%**	**99.4%**
2	DNN	GloVe	9 Projects	37.7%	14.5%	16.5%	63.5%	10.2%	78.6%	5.1%	97.2%
3	DNN	FastText	9 Projects	23.8%	9.1%	16.8%	64.5%	10.4%	79.9%	5.0%	96.9%
4	DNN	GloVe_Pre	9 Projects	**39.3%**	**15.1%**	15.6%	60.1%	10.4%	79.9%	5.0%	95.6%
5	Text-CNN	Word2Vec	9 Projects	85.3%	32.7%	21.3%	81.8%	11.7%	89.6%	5.1%	98.7%
6	Text-CNN	GloVe	9 Projects	**86.1%**	**33.0%**	**21.7%**	**83.3%**	**11.9%**	**91.2%**	**5.2%**	**99.1%**
7	Text-CNN	FastText	9 Projects	81.2%	31.1%	21.3%	81.8%	11.6%	89.3%	5.1%	98.4%
8	Text-CNN	GloVe_Pre	9 Projects	83.6%	32.1%	21.1%	81.1%	11.8%	90.9%	5.0%	96.9%
9	Bi-LSTM	Word2Vec	9 Projects	**86.9%**	**33.3%**	21.3%	82.1%	12.1%	93.1%	5.1%	98.7%
10	Bi-LSTM	GloVe	9 Projects	82.0%	31.5%	20.5%	78.9%	11.5%	88.4%	5.2%	99.4%
11	Bi-LSTM	FastText	9 Projects	86.1%	33.0%	**22.1%**	**84.9%**	**12.2%**	**93.7%**	**5.2%**	**99.7%**
12	Bi-LSTM	GloVe_Pre	9 Projects	74.6%	28.6%	20.3%	78.0%	11.6%	89.3%	5.1%	98.4%
13	Bi-LSTM_HAN	Word2Vec	9 Projects	69.7%	26.7%	**21.2%**	**81.5%**	**11.9%**	**91.2%**	5.1%	98.1%
14	Bi-LSTM_HAN	GloVe	9 Projects	74.6%	28.6%	19.8%	76.1%	11.3%	87.1%	5.1%	98.4%
15	Bi-LSTM_HAN	FastText	9 Projects	**77.1%**	**29.6%**	19.7%	75.8%	11.7%	89.6%	**5.1%**	**98.7%**
16	Bi-LSTM_HAN	GloVe_Pre	9 Projects	66.4%	25.5%	20.0%	76.7%	11.7%	89.9%	5.0%	96.9%

the metrics applied for evaluating our classifiers are the ranked retrieval precision and recall ($P@K\%$ and $R@K\%$). Moreover, our approach aimed for the retrieval task of vulnerable function, these metrics are well recommended for this task and would be more appropriate for evaluating the detection results [16]. In detail, when a detector finished testing, it produced a ranked list of functions by sorting the vulnerability probability. Among the top k percent of the total retrieved functions, we have $TP@k\%$ stands for the number of the truly vulnerable samples, while $FP@k\%$ denotes the false vulnerable ones. Next, $FN@k\%$ denotes the number of the truly vulnerable functions that could not be discovered when retrieving the top $k\%$ highest vulnerable probability. Accordingly, $P@K\%$ and $R@K\%$ are then calculated as in Table 2.

5 RESULTS

5.1 Evaluation of the Vulnerability Detectors on the Nine-projects dataset

Table 3 indicates the detection results of all trained models on the test set of the Nine-projects dataset. Specifically, the first four rows

Table 4: Distribution of precision and recall over top $k\%$ retrieved functions among the four models tested on the SARD dataset: (1-4) DNN, (5-8) Text-CNN, (9-12) Bi-LSTM, (13-16) Bi-LSTM HAN. The highest values for each model are in bold.

Index	Model	Embedding	Dataset	Precision and Recall calculated for top k% retrieved functions							
				1%		10%		20%		50%	
				P_k%	R_k%	P_k%	R_k%	P_k%	R_k%	P_k%	R_k%
1	DNN	Word2Vec	SARD	**79.3%**	**3.6%**	**59.7%**	**27.0%**	**51.8%**	**46.9%**	**41.3%**	**93.4%**
2	DNN	GloVe	SARD	37.3%	1.7%	42.7%	19.3%	40.8%	36.9%	38.2%	86.4%
3	DNN	FastText	SARD	7.3%	0.3%	8.3%	3.7%	16.1%	14.5%	29.3%	66.1%
4	DNN	GloVe_Pre	SARD	46.7%	2.1%	43.9%	19.8%	42.4%	38.4%	39.4%	89.0%
5	Text-CNN	Word2Vec	SARD	**100.0%**	**4.5%**	**100.0%**	**45.2%**	**87.9%**	**79.5%**	**44.2%**	**99.9%**
6	Text-CNN	GloVe	SARD	100.0%	4.5%	100.0%	45.2%	78.2%	70.7%	43.5%	98.2%
7	Text-CNN	FastText	SARD	100.0%	4.5%	100.0%	45.2%	86.1%	77.9%	43.6%	98.6%
8	Text-CNN	GloVe_Pre	SARD	100.0%	4.5%	100.0%	45.2%	86.9%	78.6%	44.1%	99.6%
9	Bi-LSTM	Word2Vec	SARD	**100.0%**	**4.5%**	**100.0%**	**45.2%**	**89.1%**	**80.6%**	**44.2%**	**100.0%**
10	Bi-LSTM	GloVe	SARD	100.0%	4.5%	100.0%	45.2%	88.9%	80.4%	44.2%	100.0%
11	Bi-LSTM	FastText	SARD	**100.0%**	**4.5%**	**100.0%**	**45.2%**	**89.1%**	**80.6%**	**44.2%**	**100.0%**
12	Bi-LSTM	GloVe_Pre	SARD	100.0%	4.5%	100.0%	45.2%	88.7%	80.2%	44.2%	100.0%
13	Bi-LSTM_HAN	Word2Vec	SARD	100.0%	4.5%	100.0%	45.2%	88.6%	80.1%	44.2%	99.9%
14	Bi-LSTM_HAN	GloVe	SARD	100.0%	4.5%	100.0%	45.2%	88.8%	80.3%	44.2%	100.0%
15	Bi-LSTM_HAN	FastText	SARD	**100.0%**	**4.5%**	**100.0%**	**45.2%**	**88.9%**	**80.4%**	**44.2%**	**100.0%**
16	Bi-LSTM_HAN	GloVe_Pre	SARD	100.0%	4.5%	99.9%	45.2%	88.7%	80.2%	44.2%	100.0%

summarize the performance of the DNN models trained on four embedding methods. The DNN model trained on the GlovePre embedding technique achieved the highest precision and recall when retrieving the top 1% of the vulnerable samples. However, the DNN model that used Word2Vec achieved the highest precision and recall for the other categories of Top $k\%$ retrieved functions. At the top 1% of the retrieved functions, the DNN applied GloVe model presented better results than the one using FastText. Nonetheless, when retrieving more than 10% vulnerable files, there are no significant differences in the performance among the models that used GloVe, FastText and GlovePre. Among the trained Text-CNN models, the model combined with the GloVe embedding techniques got the highest detention rates for all the top $k\%$ items (Table 3, Index 5 - 9). This is followed by the model which employed the Word2Vec method. Additionally, the Text-CNN models that used FastText, and GlovePre did not show clearly which one is better than the other and had the lowest detection rates.

Table 3 (Index 9 - 12) illustrates the detection performance of the Bi-LSTM detectors trained on four embedding methods. The Bi-LSTM models trained on Word2Vec and FastText showed quite

identical rates at Top 1% retrieved functions. Their precision rates were 86.9% and 86.1% respectively, and these are the highest detection rates among all of the trained deep models. For the rest of the Top k% items, using FastText achieved better results than using Word2Vec. The lowest testing performance was the model that applied GloVe, followed by the one that applied GlovePre. In the group of Bi-LSTM HAN models, the model which employed FastText had the highest precision and recall rates at Top 1% and Top 50%. When retrieving 10% and 20% of the detector which used Word2Vec got the best performance. The Bi-LSTM HAN models that combined with GloVe and GlovePre had the most insufficient detection results.

Overall, the Bi-LSTM detectors that applied FastText achieved the highest performance on the Nine-projects dataset at Top 10%, Top 20%, and Top 50% categories. The Text-CNN classifiers generally perform better than the Bi-LSTM with the attention mechanism. The DNN classifiers had the lowest detection results when comparing to the other classifiers.The DNN model is a generic structure and not specifically designed for processing sequential data nor for spatial data. This explain why the DNN perform poorly with the sequential code data. Given these points, our results clearly showed that both changing the neural network structures and the embedding techniques could impact noticeably to the detection results on the real-world vulnerability dataset.

5.2 Evaluation of the Vulnerability Detectors on the SARD dataset

For the model test results on the SARD dataset, the detectors that employed the DNN model showed the most inefficient detection rates. To be more specific, Table 4 (Index 1 - 4) indicates that the DNN model combined with the Word2Vec embedding method produced the highest detection rates. This is followed by the models that used GloVe and GlovePre respectively.

The DNN model which employed FastText had the lowest detection rates. In fact, this result is quite identical to the DNN models which were trained on the Nine-projects dataset. Among the trained Text-CNN models, the model using Word2Vec achieved the most effective performance, while the one that employed GloVe was the least effective model. When retrieving less than 20% of the vulnerable samples, all of the Text-CNN models could retrieve 100% of the vulnerable files.

In addition, Table 4 (Index 9 - 16) shows that the detection rates at Top 1%, Top 10% and Top 50 % of the vulnerable samples were similar between the Bi-LSTM and Bi-LSTM HAN models. In this case, the results of the two models also showed that changing the embedding methods does not affect the performance. Owing to the fact that the SARD dataset had a well-balanced rate between vulnerable and non-vulnerable samples, the detectors could be trained sufficiently. Particularly, all the detectors can also reach their highest precision at Top 1% and Top 10% categories like in the case of Text-CNN.

In summary, the detectors that employed Bi-LSTM and Bi-LSTM HAN performed better than those used Text-CNN on average. This can be explained since Bi-LSTM model is well-known for dealing with sequential data. Furthermore, the bidirectional characteristic of Bi-LSTM allowed the model to learn effectively the sequence data from both forward and backward directions. Our results showed that changing the embedding techniques still affects the detection performance in the case of the DNN and the Text-CNN models. Meanwhile, the Bi-LSTM and Bi-LSTM HAN detectors showed no significant differences in detection performance regardless of using different embedding methods.

6 CONCLUSION AND FUTURE WORK

This paper presented an approach to compare the effectiveness of four word embedding methods for several neural networks. The system trained the models and tested them on two genres of function level datasets written in the C programming language. Training the detectors on the SARD dataset required more time since its size is much larger than the other dataset. Among all of the considered embedding techniques, FastText achieved the shortest time consumption for training the vulnerability detectors, while training with GloVe consumed the longest time.

With the real-world implemented dataset, the models showed differences clearly in detection performance. In contrast, the trained some models such as the Bi-LSTM models could present sufficient but identical vulnerability retrieval results. This is due to the vulnerability patterns extracted from the artificially synthesized samples that are much simpler to capture compared to the real-world samples by the neural networks. This is worth noticing since the real vulnerable dataset in the released software code can be limited to size and numbers in varied scenarios. Thus, it is vital to choose the effective combination of embedding techniques and neural network structures to build an effective detection system that can adapt well to the dataset.

Our work evaluated the use of word embedding algorithms for four different neural networks, and the system can train the vulnerability detectors at the function level. It can assist users to explore the good combinations of embedding methods and deep learning models to build effective vulnerability detection systems. In the future, we can extend our work in several ways to improve system performance. First, we can collect and build up the volume of the real vulnerable dataset to resolve the imbalance issue in the open-source dataset. Second, we improve the input embedding layer for the neural networks by using a Lexer to build an intermediate representation of the code. This could use a much smaller vocabulary size and consequently reduce the dimensionality of the training examples. Finally, more complex neural network models should be experimented with to reduce the gap between natural language text and programming. This would allow the vulnerability detection system to learn better and accommodate to other programming languages.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 265–283.

[2] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.

[3] Paul E Black. 2018. *Juliet 1.3 Test Suite: Changes From 1.2.* US Department of Commerce, National Institute of Standards and Technology.

[4] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics* 5 (2017), 135–146.

[5] CVE. 2019. Common Vulnerabilities and Exposures website. https://cve.mitre.org/.

[6] Guisheng Fan, Xuyang Diao, Huiqun Yu, Kang Yang, and Liqiong Chen. 2019. Software defect prediction via attention-based recurrent neural network. *Scientific Programming* 2019 (2019).

[7] Yong Fang, Yongcheng Liu, Cheng Huang, and Liang Liu. 2020. FastEmbed: Predicting vulnerability exploitation possibility based on ensemble machine learning algorithm. *Plos one* 15, 2 (2020), e0228439.

[8] Jacob A Harer, Louis Y Kim, Rebecca L Russell, Onur Ozdemir, Leonard R Kosta, Akshay Rangamani, Lei H Hamilton, Gabriel I Centeno, Jonathan R Key, Paul M Ellingwood, et al. 2018. Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497* (2018).

[9] Jordan Henkel, Shuvendu K Lahiri, Ben Liblit, and Thomas Reps. 2018. Code vectors: understanding programs through embedded abstracted symbolic traces. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 163–174.

[10] Yoon Kim. 2014. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882* (2014).

[11] Maciej Kula. 2019. A python implementation of GloVe: glove-python. https://github.com/maciejkula/glove-python.

[12] Zhen Li, Deqing Zou, Jing Tang, Zhihao Zhang, Mingqian Sun, and Hai Jin. 2019. A comparative study of deep learning-based vulnerability detection system. *IEEE Access* 7 (2019), 103184–103197.

[13] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).

[14] Guanjun Lin, Wei Xiao, Jun Zhang, and Yang Xiang. 2019. Deep Learning-Based Vulnerable Function Detection: A Benchmark. In *International Conference on Information and Communications Security*. Springer, 219–232.

[15] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, Olivier De Vel, Paul Montague, and Yang Xiang. 2019. Software vulnerability discovery via learning multi-domain knowledge bases. *IEEE Transactions on Dependable and Secure Computing* (2019).

[16] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. 2009. *Introduction to Information Retrieval*. Cambridge University Press.

[17] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[18] Serguei A Mokhov, Joey Paquet, and Mourad Debbabi. 2014. The use of NLP techniques in static code analysis to detect weaknesses and vulnerabilities. In *Canadian Conference on Artificial Intelligence*. Springer, 326–332.

[19] NSCLab. 2020. Cyber Code Intelligence GitHub website. https://github.com/cybercodeintelligence/CyberCI.

[20] NVD. 2019. National Vulnerability Database website. https://nvd.nist.gov/.

[21] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 1532–1543.

[22] Michael Pradel and Koushik Sen. 2017. Deep learning to find bugs. *TU Darmstadt, Department of Computer Science* (2017).

[23] Achyudh Ram, Ji Xin, Meiyappan Nagappan, Yaoliang Yu, Rocío Cabrera Lozoya, Antonino Sabetta, and Jimmy Lin. 2019. Exploiting Token and Path-based Representations of Code for Identifying Security-Relevant Commits. *arXiv preprint arXiv:1911.07620* (2019).

[24] Radim Řehůřek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. ELRA, Valletta, Malta, 45–50.

[25] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 757–762.

[26] SARD. 2019. Software Assurance Reference Dataset project. https://samate.nist.gov/SRD/.

[27] Jürgen Schmidhuber and Sepp Hochreiter. 1997. Long short-term memory. *Neural Comput* 9, 8 (1997), 1735–1780.