

# The Comparison of Word Embedding Techniques in RNNs for Vulnerability Detection

Hai Ngoc Nguyen<sup>1</sup>, Songpon Teerakanok<sup>2</sup><sup>a</sup>, Atsuo Inomata<sup>3</sup> and Tetsutaro Uehara<sup>1</sup><sup>b</sup>

<sup>1</sup>*Cyber Security Lab, College of Information Science and Engineering, Ritsumeikan University, Japan*

<sup>2</sup>*Research Organization of Science and Technology, Ritsumeikan University, Japan*

<sup>3</sup>*Graduate School of Information Science and Technology, Osaka University, Japan*

**Keywords:** Deep Learning, Word Embeddings, Vulnerability Detection, RNNs.


**Abstract:** Many studies have combined Deep Learning and Natural Language Processing (NLP) techniques in security systems in performing tasks such as bug detection, vulnerability prediction, or classification. Most of these works relied on NLP embedding methods to generate input vectors for the deep learning models. However, there are many existing embedding methods to encode software text files into vectors, and the structures of neural networks are immense and heuristic. This leads to a challenge for the researcher to choose the appropriate combination of embedding techniques and the model structure for training the vulnerability detection classifiers. For this task, we propose a system to investigate the use of four popular word embedding techniques combined with four different recurrent neural networks (RNNs), including both bidirectional RNNs (BRNNs) and unidirectional RNNs. We trained and evaluated the models by using two types of vulnerable function datasets written in C code. Our results showed that the FastText embedding technique combined with BRNNs produced the most efficient detection rate, compared to other combinations, on a real-world but not on an artificially-produced dataset. Further experiments on other datasets are necessary to confirm this result.


## 1 INTRODUCTION

Software quality is a significant concern within the cybersecurity field since vulnerabilities in software code can greatly damage an organization's day-to-day operations. As a matter of fact, securing the software code by both dynamic and static analysis methods has been studied widely among security experts. In software source code, many similar characteristics were present in natural language texts (Allamanis et al., 2018). For that reason, the use of NLP applications in automatically detecting vulnerability in code has been investigated. With the recent breakthrough of deep learning in numerous fields including NLP applications, researches have shown the great potential of deep learning in source code static analysis (Russell et al., 2018). In any machine learning or deep learning system, a specified embedding technique is required for generating model inputs as vector representations. Nevertheless, there are many existing embedding methods in the NLP field such as Word2Vec

(Mikolov et al., 2013) and GloVe (Pennington et al., 2014). This makes it difficult to select a suitable method for the vector encoding tasks.

Among deep learning models, sequence models like RNNs are famous for dealing with text sequences. The simple RNN model faces problems of gradients vanishing or exploding when the input sequences get too long. To deal with long sequence inputs, other structures of RNNs, namely Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997) and Gated Recurrent Units (GRUs) (Kostadinov, 2017), were introduced. These sequence models proved to be the suitable learning models for encoding code files in Li (2018) and Li (2019). These studies used Word2Vec to produce code vector representations, but other embedding methods like GloVe or FastText (Bojanowski et al., 2017) have yet to be evaluated on these models. Different combinations of the embedding method and deep learning model can capture different types of knowledge representations like linguistic contexts of identifiers and their temporal sequences. Changing the embedding method for training the deep model could therefore impact the performance of classifiers. Moreover, the deep

<sup>a</sup>  <https://orcid.org/0000-0002-1058-149X>

<sup>b</sup>  <https://orcid.org/0000-0002-8233-130X>

model also requires distinguished amounts of time consumed for training and testing on different types of representations. To select the suitable embedding method can be a critical task since it can affect the performance of the models and the time complexity of training and detection. Thus, we aim to determine the most viable combination between certain sequence models and available embedding methods for generating semantic vectors.

We present a system to train code vulnerability detectors for evaluating four word embedding methods combined with four popular RNNs. The system was built based on the open-source API benchmark (Lin et al., 2019a). As an extended look to the API, we also used two types of the dataset which were originally setup as the baselines for comparisons in the benchmark. Both datasets contain files written in C program language where each file represents either a vulnerable or non-vulnerable function. The first dataset is the Nine-projects dataset that was constructed from nine open-source projects with the vulnerability information extracted from the National Vulnerability Database (NVD, 2019) and the Common Vulnerabilities and Exposures (CVE, 2019) websites. The second dataset is obtained from the Software Assurance Reference Dataset (SARD, 2019) project, which consists of the artificially synthesized function files. Through our experiments, we explored the combinations of the word embeddings techniques and RNNs for building vulnerability detectors at the function level. Our system trained and tested the vulnerability detectors in a supervised manner of deep learning. Since the system processes program source code as text files in file-level classification exercises, source code analysis is not necessary to analyze the program.

The main contributions of this paper are concluded as follows:

- We extend a benchmark system by evaluating three additional word embedding techniques to encode the C program functions as vector representations.
- We implement the LSTM, bidirectional LSTM (Bi-LSTM), GRU, and bidirectional GRU (Bi-GRU) models for training the vulnerability detectors at the function file level on two different datasets.
- We conduct an overall performance evaluation of all trained classifiers on the two datasets. Particularly, each classifier is examined on different input representations to discover the compatibility of the embedding algorithms and the models.

The rest of this paper is arranged as follows: Section 2 presents the related studies where the word em-

bedding techniques and 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

Word embedding techniques are widely used in building NLP applications. Inspired by the success of NLP and neural language models, the earlier studies observed the strong resemblances in semantic and syntactic information between natural language to the programming language. They had leveraged the advantages of these methods to detect vulnerabilities and predict defects in software code analysis. One of the earliest applications of this technique was done by implementing classical NLP algorithms, such as n-grams, combined with machine learning techniques for non-NLP tasks of detecting and classifying vulnerable code practices in programming languages (Mokhov et al., 2014). It was done in an identical manner as a classic text identification task.

Afterwards, more studies tested increasingly complicated machine learning models while employing different word embedding techniques for generating vector representations as inputs for the training process. Pradel and Sen used Word2Vec for generating code vectors derived from the custom Abstract Syntax Trees (ASTs) - based contexts (Pradel and Sen, 2017). These vectors were used to train deep learning models to detect vulnerability in JavaScript code. Likewise, the Word2Vec model was applied for making vector representations from C/C++ source code and trained vulnerability detection models with both Word2Vec representations and the control flow graphs (CFGs) data (Harer et al., 2018). Instead of using Word2Vec, Henkel applied the GloVe model to produce vectors learned from the Abstracted Symbolic Traces of C programs (Henkel et al., 2018). Furthermore, FastText was used in FastEmbed for vulnerability prediction based on ensemble machine learning models (Fang et al., 2020). Although there are already several examples of using word embedding techniques in vulnerability detection, comparisons between these techniques were not possible to make due to differences in baseline dataset types and machine learning models structures.

Deep learning has recently attracted more interest in code analysis research since it has achieved great success in numerous fields such as computer vision, image processing, and natural language pro-

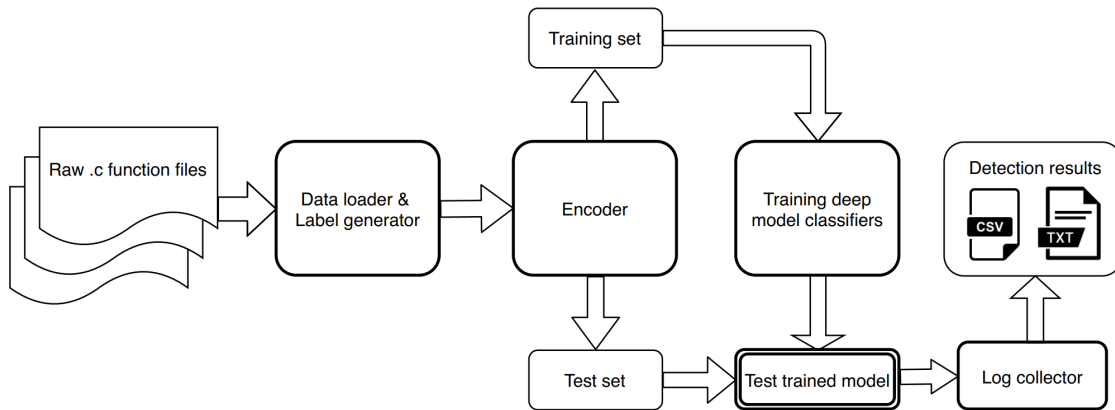


Figure 1: Approach Overview.

cessing. By converting the self-constructed dataset called code gadget into Word2Vec vector representations, VulDeePecker was developed from the Bi-LSTM model to detect specific types of C/C++ vulnerabilities (Li et al., 2018). The same authors also provided a comparison for several deep learning models on the same artificially constructed dataset (Li et al., 2019). Another study used Word2Vec for the embedding task, but their model architecture employed a convolutional layer on top of the standard Bi-LSTM model (Niu et al., 2020). Although the mentioned systems have achieved well vulnerability detection performance, their trained models were tested on their self-constructed datasets like building ASTs, CFGs, etc. The success of the methods based on syntactic artifacts dataset raises a question of whether the customized dataset proved more useful than basic input such as the word vectors. This is challenging for making an overall comparison between these systems and requires program analyzing expertise.

There are studies that have started to explore the effectiveness of using different representations for deep learning models to deal with program classification tasks. A comparative analysis was conducted to assess how different deep learning models learn over distinctive input representations of Java code (Ram et al., 2019). Additionally, Lin (2019) proposed a benchmark framework and compared three models which are Text-CNN (Kim, 2014), DNN and LSTM. However, further evaluations of using different embedding algorithms or different neural networks are yet to be explored. In this paper, we present an approach that allows users to observe the performance of deep learning models on different types of vector representations. The detection granularity of this project is at the function level based on the two types of datasets.

### 3 APPROACH

Our goal is to design a system for investigating the effectiveness of word embedding techniques for training vulnerability detectors. The system initially loads the source code files and preprocesses them into sequences of word tokens which were identifiers, data types, variables, etc. Each of these sequences stands for a semantic function representation. The list of representations would have the corresponding list of labels by processing the function names. Subsequently, the system applies the predefined word embedding algorithm to map the sequences of tokens into vector representations. The specified neural network used eighty percent of the vector representations for training the vulnerability detector, while the rest of the representations are tested by the trained detector. After testing, the vulnerable probabilities of the test samples were produced correspondingly.

#### 3.1 Overview

In this work, we apply four popular word embedding techniques to train four different RNNs. Figure 1 presents our workflow. In detail, given a corpus of function code files, training a deep model classifier includes several steps. In the first stage, the source code files are loaded and processed to generate sequence data and labels. The data is then passed to the next stage to be transformed into the code embedding vectors. These vectors will then be partitioned and fed to the constructed neural network for the training process. When training is completed, the model is tested, and the detailed logs are automatically collected in the last phase.

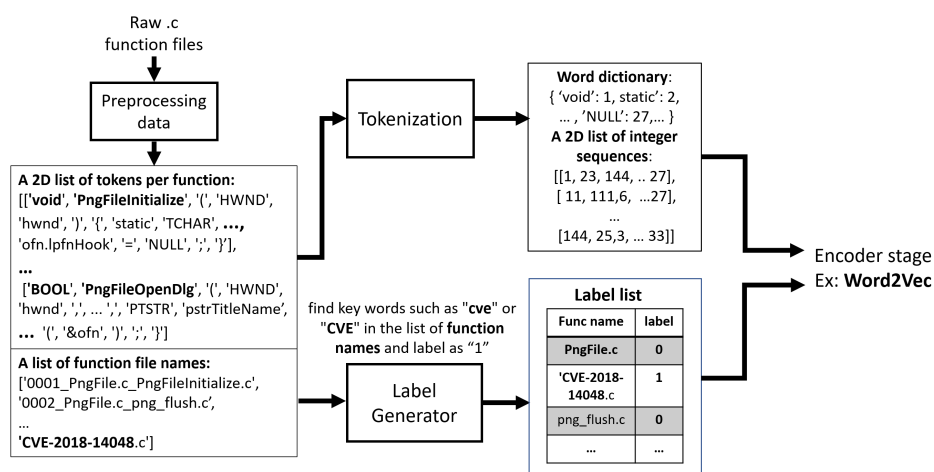


Figure 2: Data Loader and Label Generator Module.

### 3.2 Data Loader and Label Generator

Figure 2 shows the data flow within the Data Loader and Label Generator module. This module initially loads the source code files to get a list of identifier tokens and a list of function names. When preprocessing the raw data, each file in the corpus was split into a list of words and punctuation characters before performing tokenization technique by Keras tokenizer (Chollet et al., 2015). By fitting the whole corpus data to the tokenizer, it turns each function file into a sequence of integers. Finally, the list of these sequences and the vulnerability labels are produced for the latter encoding stage.

To generate ground truth labels, we loaded two datasets into our module and tasked the module to detect certain keywords in the filenames. Files which contained the designated keywords were then labeled as either vulnerable (1) or invulnerable (0). These two datasets are the Nine-projects and the synthetic SARD datasets. For the Nine-projects dataset, we set the vulnerable keywords to match the strings such as “CVE” or “cve”. Similarly, the SARD’s files contain such keywords as “BAD”, “bad”, etc. These keywords were incorporated directly into the module. One of the module settings is to select the type of dataset before execution. This guarantees the system adaptability to other types of dataset.

Here, the label generator settings can be customized to a suitable type of dataset. It is important to notice that the function body of the files in the SARD dataset also has such keywords as those we picked for labeling. These words potentially add bias to the training process of the deep learning models. Since word embedding techniques are used for generating vectors, the model can look at the keyword vectors to decide the vulnerability results. Therefore, we scan

the function body to look for those keywords, replacing them with the same length dummy strings. Hence, those words in the function body can not affect the model performance.

### 3.3 Encoder Module

This module converts each code function file into a vector that could retain both semantic and syntactic information out of the source code. To allow the models to learn effectively in the later stage, it is important to extract the information from the code tokens. Particularly, for preserving the semantic knowledge expressed by the identifier names, we used the embedding layer to map these identifiers in code to the semantic vector representation. By observing the content of the two datasets, more than 90% of sequences were acknowledged to have lengths that are shorter than 1000. To balance the sequence length and the sparsity of sequences (Lin et al., 2019b), we select the maximum length of code sequence to 1000. For those functions containing the sequence length longer than 1000, they are truncated to length 1000. Conversely, we append zeros at the end for the sequences having a shorter length than 1000. Next, the module will pick one of the embedding methods to start mapping the file sequences into a meaningful code vector. Word2Vec, GloVe, FastText and GloVe pre-trained models (GloVePre) (Pennington et al., 2014) were implemented in our work. We set the embedding layer to generate fixed-length vectors at the dimension ( $d = 100$ ) as default.

The Word2Vec model was implemented to learn semantic information from a large amount of raw data. The model was provided by the GenSim (Řehůřek and Sojka, 2010) package with the Continuous Bag of Words (CBOW) and Skip-gram algo-

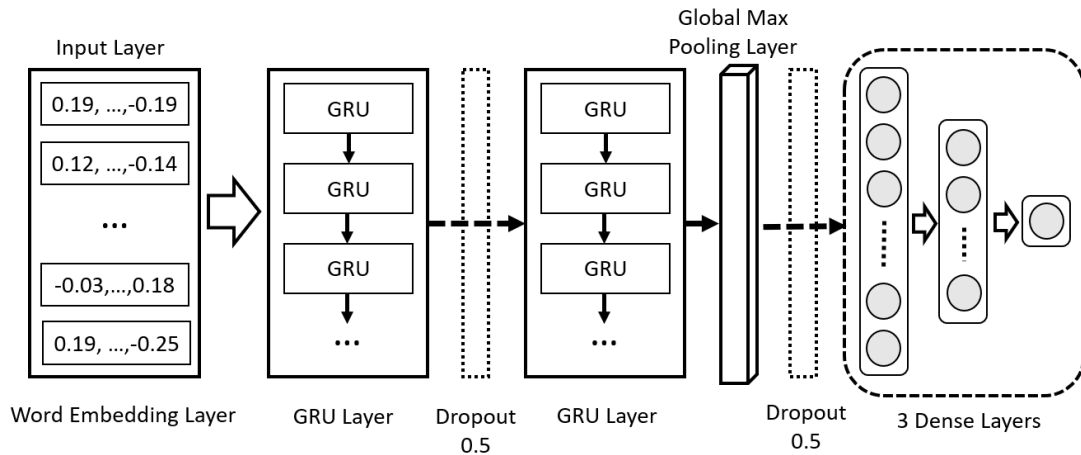


Figure 3: The example of the GRU model structure.

rithms. CBOW learns to predict the word by its context, while the Skip-gram is built to predict the context. Therefore, we chose CBOW over the other since we needed to extract the syntactic code sequence information but not its context. The other parameters of the Word2Vec were setup by default.

In a similar manner, the FastText model was constructed from the GenSim package. Moreover, it is considered as the main comparison to the Word2Vec model in training the neural networks, since FastText can even construct the vector for the word from its character  $n$  grams even when the word is out of its vocabulary. The number of threads and the *window* size were customized to 4 and 5 likes in the Word2Vec model, while the rest of the parameters were configured by default.

Finally, the GloVe model was also trained from the given corpus of code. Constructing GloVe was done by the glove-python package (Kula, 2019). We tuned the GloVe model with the learning rate at 0.05, set the *window* parameter to 10, and trained it with four threads in 500 Epochs. While this generates embeddings, which is an identical task to Word2Vec, GloVe presented its embeddings by factorizing the logarithm of the corpus word co-occurrence matrix. The GloVePre model was additionally 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.

Given these points, the sequences collected from the previous stages can be translated into vector representations with the shape of  $(1000, 100)$  by one of the embedding models. Here, the selected model is responsible for preserving the code semantic information into the embeddings.

### 3.4 Training Module

Taking the meaningful code representation vectors from the previous stage as inputs, the training module is responsible for training the neural networks to distinguish between vulnerable and non-vulnerable function samples. Our work focuses on investigating the effectiveness of RNNs since such models like Long Short-Term Memory (LSTM) or Gated Recurrent Unit (GRU) are well-known for dealing with sequential data (Li et al., 2019). More importantly, their bidirectional forms, having a backward layer and a forward layer can adapt efficiently with program code, where the order of statements plays a significant role in vulnerability detection. The module configuration can be customized to select one of the four RNNs models. Here, we explored LSTM, Bi-LSTM, GRU, and Bi-GRU for learning the features extracted from the embedded code representations.

The LSTM network in this work was designed with eight layers. The first layer is an LSTM recurrent layer with the 128 neurons. This layer takes the meaningful embedding vectors extracted from code sequences in the encoder module as input. The second layer is a dropout regularization layer with dropout rate at 0.5. It helps the model to prevent over-fitting issues by randomly removing hidden units and their connections in the training process. The third layer is another LSTM recurrent layer with 128 neurons. After this, the output of the LSTM layer was concatenated for downsampling by a pooling layer. Another dropout layer with the same rate is added after the pooling layer. The last three layers are dense layers. The first dense layer has 64 neurons, and this number of neurons is reduced by half in the second dense layer. The last layer has only one neuron and uses sigmoid activation for converging the output into a single

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

probability between 0 and 1. The GRU network was constructed in the same way as the LSTM network. The only difference is that GRU recurrent layers were implemented instead of using LSTM layers. The example of the GRU model structure is described in Figure 3.

The BRNNs were constructed in a similar structure for both Bi-LSTM and Bi-GRU. The Bi-LSTM model consists of eight layers. The first layer is a bidirectional LSTM recurrent layer with 64 LSTM cells. The bidirectional layer allows its output to concurrently acquire the information from both preceding and succeeding scenarios. The second layer is a dropout regulation layer with the same dropout rate as in LSTM model. The third layer is designed the same as the first layer. Henceforth, the output of the Bi-LSTM layer will be reduced by one dimension with a pooling layer. After pooling, a dropout layer using the same rate is used. Ultimately, the last three dense layers were built in the same way as in the structure of the unidirectional RNNs. Regarding the network structures and hyper-parameters, the Bi-LSTM model was constructed based on the work of the group of Lin (2019b). Following that reference, we designed and tuned the other network architectures using similar methodology.

### 3.5 Test Module and Logs Collector

In the previous stage, the constructed model employed eighty percent of the dataset for training the classifier. Here, the test module can select one of the trained classifiers by specifying its name and use the rest of the dataset for testing. When the test process is completed, the raw CSV data will be presented as a list of function names and its vulnerability probability. Next, the logs collector processes the CSV file data and sorts the list of function names to another table following their vulnerability probability in the order from the highest to the lowest. Finally, it can calculate performance metrics and produce results as a .txt file.

## 4 EXPERIMENTS

### 4.1 Experimental Setup

Our experiments focused on answering the following research questions:

- Question 1: Can applying different embedding methods improve the effectiveness of the vulnerability detector?
- Question 2: How are the training speed and performance of each model when using different embedding techniques?
- Question 3: Would change the neural network model affect detection performance?

To summarize, we used four embedding methods to train and test four types of RNNs on two types of datasets respectively. It means that we had trained and tested 16 classifiers for each dataset. We set the optimizer for all the networks as Stochastic Gradient Descent (SGD) followed by the default setting of Keras. The binary cross-entropy was selected as our loss function. 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) (Abadi et al., 2016). 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 (Kula, 2019). 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

The Nine-projects dataset is the proposed dataset in the benchmark API (Lin et al., 2019a). The authors had shared their dataset on their GitHub website (NSCLab, 2020). The second dataset is the synthetic dataset supplied by the Software Assurance Reference Dataset project (SARD, 2019). The project is

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.

known as the Juliet Test Suites (Black, 2018). It includes test functions for C/C++ and Java. In this work, we only take the C source code for our experiments. Following the studied cases in the benchmark API, we extracted randomly 35000 vulnerable and 40000 non-vulnerable C function files from the SARD functions dataset provided by the same GitHub repository. For both datasets, after being encoded to the labeled vectors, the dataset is distributed with the rate of 0.8 for training and validation set, and 0.2 for the test set. The content of the datasets is described in Table 1. We keep this data partition setting to train and test all the deep learning models.

### 4.3 Performance Metrics

For most of the cases, precision, recall, and F1-score are used for evaluating deep learning classification models. However, in many circumstances of vulnerability detection, the dataset imbalance between non-vulnerabilities and vulnerabilities showed that these metrics would undervalue the model detection performance (Lin et al., 2019a). Therefore, the metrics applied for evaluating our classifiers are the ranked retrieval precision and recall ( $P@K\%$  and  $R@K\%$ ). Moreover, our approach aims for the retrieval task of vulnerable function, these metrics are well recommended for this task and would be more suitable for evaluating the detection results (Manning et al., 2009).

Specifically, when a detector finishes testing, it will produce 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. For instance, the total number of test functions was 60000, and the number of vulnerable functions was 1500. With  $k = 10$ , the top 10% will accordingly retrieve 6000 files with the highest vulnerable rate. Furthermore, given the following details: 1400 vulnerable files were found true positive among the 6000

files, 100 vulnerable files were found to be missing, and 4500 non-vulnerable files were found to be false positive; the reported values will be as follows:  $TP@k\%$  is 1400,  $FP@k\%$  is 4500, and  $FN@k\%$  is 100. Hence,  $P@K\%$  and  $R@K\%$  can be calculated as the formulas in Table 2.

## 5 RESULTS

### 5.1 Model Training Results

The time consumed for the training neural models usually receives less attention than standard performance metrics like precision, recall rate, etc. However, when the models reach their capacity and show identical results in performance, understanding time complexity would be a valuable insight to choose the appropriate methods for training. Figure 4 summarizes the training time of the four RNNs on the SARD and the Nine-projects datasets. Training the detectors on the SARD dataset took more time since its size is much larger than the other. In general, using FastText achieves the shortest time consumption for training the classifiers, while training with GloVe takes the longest time. Training models by Word2Vec is faster than by GloVePre. Indeed, this was expected since FastText was proved to accomplish optimal speed when compared to other word embedding models (Joulin et al., 2016). Among the neural networks, the GRU and Bi-GRU models require the largest amount of time to train. Conversely, Bi-LSTM and LSTM models take less time for training than the others.

### 5.2 Evaluation of Word Embedding Methods on the Trained Models

In Figure 5.A, the results show the detection performance of the LSTM models trained on four embedding methods. The LSTM models which were trained on FastText’s vector representations achieved the highest precision and recall for all categories of

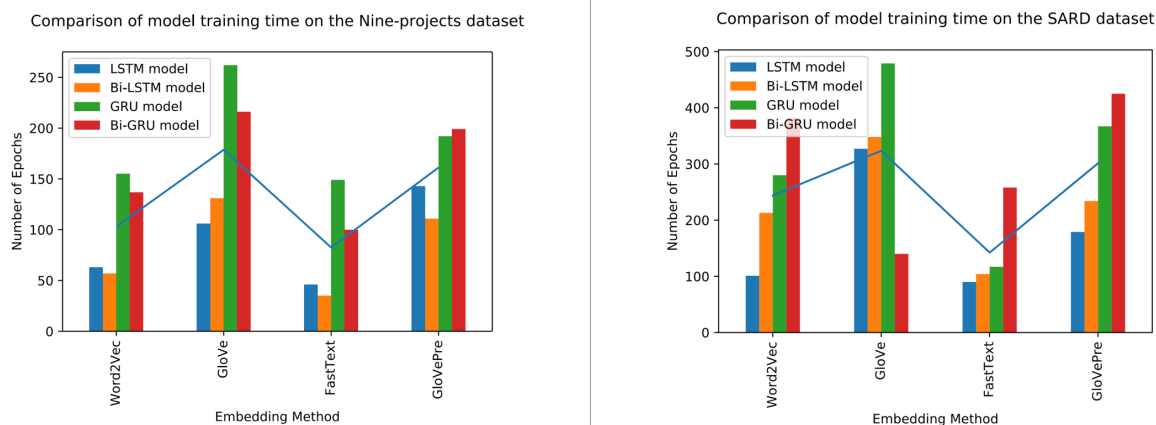


Figure 4: Time consumption summary for training RNNs on the SARD and the Nine-projects datasets.

top  $k\%$  retrieved functions. In detail, the precision at the top 1% reached 83%. The recall rate at the top 20% and top 50% could reach 93% and 99%. The next best performing detector was the model trained on GloVe, which got 80% for the precision at the top 1%. The next ranks followed by the models applied by Word2Vec, and GloVePre.

Figure 5.B presents the detection performance of the Bi-LSTM models 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 87 and 86% respectively. For the rest of the top  $k\%$  items, using FastText still achieved the highest recall rates. The lowest testing performance was the model that applied GloVe, followed by the one that applied GloVePre. Noticeably, the performance of the Bi-LSTM models is much higher than the LSTM models. For example, at the top 1% retrieved functions, the precision rate had improved by 11% in the case of Word2Vec and by 4% in the case of FastText.

When retrieving less than 50% of vulnerability functions, Figure 5.C shows a similar trend that detectors using FastText achieved the highest precision and recall rates. On average, the GRU models have lower performance than the Bi-LSTM models, but higher than the LSTM models. For instance, in the top 1% vulnerable samples for the Word2Vec category, the LSTM detector reached only 76% in precision rate while the precision rates of the GRU and Bi-LSTM models were higher by 8% and 11% respectively. The performance of the GRU models that employed GloVe and Word2Vec was similar.

In an identical manner, Figure 5.D witnesses the Bi-GRU model where FastText was implemented,

achieving the highest performance for the top 1% most vulnerable files. Compared with other models, the Bi-GRU detectors have much better performance than the LSTM and GRU detectors. Likewise, the Bi-GRU models achieved higher performance rates in both precision and recall for top  $k$  items when comparing to the Bi-LSTM models. Hence, there is a clear performance gap between the bidirectional RNNs and the unidirectional RNNs.

Figure 5.A and 5.B show, LSTM, and Bi-LSTM models performed better with FastText and Word2Vec techniques. For all the models, the precision rates at Top  $k$  become lower, and the recall rates increase when  $k$  is increasing. This is due to the proportion of the vulnerable data getting smaller when the number of the retrieved files increases. When retrieving 50% the total number of the files, all the detectors could collect more than 97% of the relevant files. Particularly, the Bi-LSTM models applied FastText, and the Bi-GRU model employed GloVe could retrieve all relevant vulnerable functions back.

Overall, the detectors that applied FastText got the best performance on the Nine-projects dataset. The models that used GloVe and Word2Vec did not clearly show which one is better than the other. GloVePre generally had the lowest general rates, that was because its embedding vectors were meant for words in the predefined dictionary. The better performance of FastText is likely due to its capability to produce the vector for a word from its character  $n$ -grams even if the word had no presence in the training corpus (Bojanowski et al., 2017). GloVe and Word2Vec does not have the same capability. Furthermore, as an extension model of Word2Vec, FastText was able to generate better embeddings for infrequent words since it



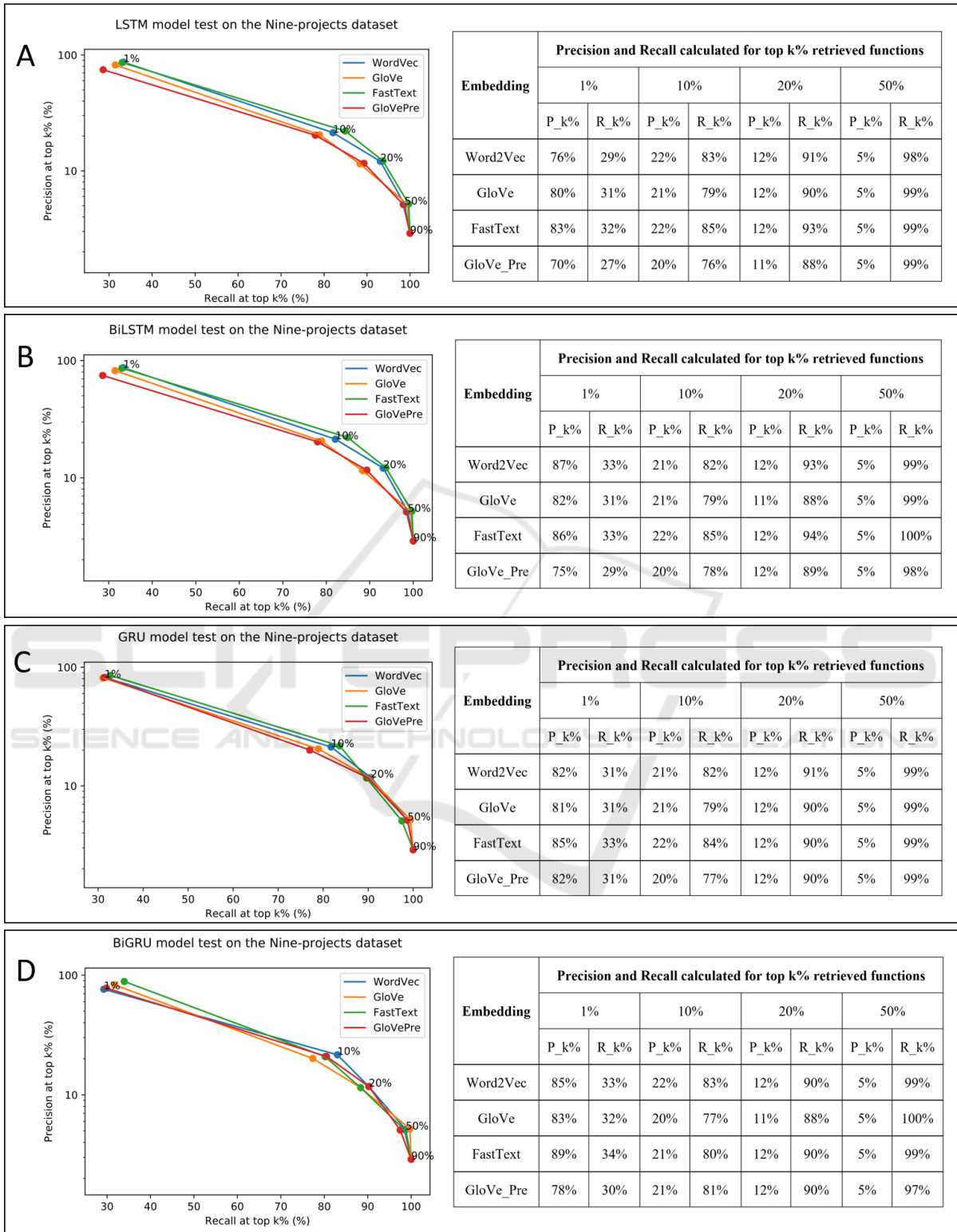


Figure 5: Distribution of precision and recall over top  $k$ % retrieved functions among the four models tested on the Nine-projects dataset: (A) LSTM, (B) Bi-LSTM, (C) GRU, (D) Bi-GRU.

Table 3: Precision and Recall over top  $k\%$  retrieved functions of the RNNs on the SARD dataset.

Index	Model	Embedding	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	LSTM	Word2Vec	100.0%	4.5%	100.0%	45.2%	88.9%	80.4%	44.2%	100.0%
2	LSTM	GloVe	100.0%	4.5%	100.0%	45.2%	89.2%	80.7%	44.2%	100.0%
3	LSTM	FastText	100.0%	4.5%	100.0%	45.2%	89.0%	80.4%	44.2%	100.0%
4	LSTM	GloVe_Pre	100.0%	4.5%	100.0%	45.2%	89.2%	80.6%	44.2%	100.0%
5	Bi-LSTM	Word2Vec	100.0%	4.5%	100.0%	45.2%	89.1%	80.6%	44.2%	100.0%
6	Bi-LSTM	GloVe	100.0%	4.5%	100.0%	45.2%	88.9%	80.4%	44.2%	100.0%
7	Bi-LSTM	FastText	100.0%	4.5%	100.0%	45.2%	89.1%	80.5%	44.2%	100.0%
8	Bi-LSTM	GloVe_Pre	100.0%	4.5%	100.0%	45.2%	89.2%	80.7%	44.2%	100.0%
9	GRU	Word2Vec	100.0%	4.5%	100.0%	45.2%	88.9%	80.4%	44.2%	100.0%
10	GRU	GloVe	100.0%	4.5%	100.0%	45.2%	88.5%	80.1%	44.2%	100.0%
11	GRU	FastText	100.0%	4.5%	100.0%	45.2%	89.2%	80.7%	44.2%	100.0%
12	GRU	GloVe_Pre	100.0%	4.5%	100.0%	45.2%	88.8%	80.3%	44.2%	100.0%
13	Bi-GRU	Word2Vec	100.0%	4.5%	100.0%	45.2%	89.0%	80.5%	44.2%	100.0%
14	Bi-GRU	GloVe	100.0%	4.5%	100.0%	45.2%	88.5%	80.0%	44.2%	100.0%
15	Bi-GRU	FastText	100.0%	4.5%	100.0%	45.2%	88.6%	80.1%	44.2%	100.0%
16	Bi-GRU	GloVe_Pre	100.0%	4.5%	100.0%	45.2%	89.3%	80.8%	44.2%	100.0%

treats each word by considering the word's character n grams. Finally, the use of the hierarchical softmax with careful implementation in FastText helps to optimize the computation process (Joulin et al., 2016).

For the model test results on the SARD dataset, we observe the precision and recall rate at top  $K$  are very identical between all the models. On Table 3, the results showed that changing the embedding methods did not greatly affect the performance in the case of the synthetic dataset, since the dataset has a well-balanced rate between vulnerable and non-vulnerable files and contains a large enough size to effectively train the detectors. All the detectors can reach their highest precision at Top 1% and Top 10%. At Top 50%, all the vulnerable functions were retrieved successfully. Our experiments confirm further the conclusion in (Lin et al., 2019a) that no statistically significant difference in performance was found for all RNNs on the SARD dataset regardless of combining different embedding techniques. The vulnerabil-

ity patterns extracted from the artificially synthesized samples are much simpler to capture compared to the real-world samples by the neural networks.

### 5.3 The Comparisons of the Four Deep Neural Networks

As Table 3 indicates, the performance of all detectors which were trained on the SARD dataset are sufficient and identical due to the dataset synthetic characteristic. Therefore, the comparisons between the four neural networks are made based solely on the detector trained on the Nine-projects dataset (Figure 5). In general, the BRNNs perform better than the unidirectional RNNs, likely due to the advantages of BRNNs discussed in section 3.4. Among BRNNs, the Bi-GRU detectors showed higher precision and recall rates in top  $k\%$  most vulnerable samples on average. Likewise, the GRU models achieved better performance than the LSTM ones in the group of the

unidirectional RNNs. The GRU and Bi-GRU models can detect more effectively than the LSTM and Bi-LSTM models in case of training on the Nine-projects dataset. This is due to the structure of the GRU networks being more compatible with the small size dataset. Since the Nine-projects dataset has smaller size than the SARD dataset, GRU models had better advantages for less memory consumption.

## 6 CONCLUSION AND FUTURE WORK

Automated detection of software vulnerability is an important direction in cybersecurity research. However, conventional techniques such as dynamic analysis or symbolic-execution showed inefficiency when dealing with an immense amount of source code (Lin et al., 2019b). To enhance the vulnerability discovery capability, applying deep learning techniques was necessary to speed up the code analysis process. Our work presented an approach to examine the effectiveness of word embeddings combined with four deep learning models for the vulnerability detection task. The system trained the models and tested them on two genres of the datasets. With the synthetic dataset, all models could present sufficient but identical vulnerability retrieval results. In contrast, the models showed differences clearly with the real-world implemented dataset. 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 select the right combination of embedding methods and neural network structures to build an effective detection system that can accommodate well to the dataset.

Our approach investigated the use of embedding algorithms on the supervised learning methods, and the system can generate the vulnerability detectors at the function level. It can be used as an assisting tool for selecting the good combinations of embedding methods and deep learning models for building effective vulnerability detection systems. There are several research directions for extending our work and improving 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 can work on implementing other embedding solutions such as adapting an ASTs extractor (Kovalenko et al., 2019). This could extract different patterns of information in source code for the center machine learning models to learn in the later stage. Finally, building better neural network mod-

els should be investigated to reduce the gap between natural language text and programming. This would allow the vulnerability detection system to learn better and adapt to other programming languages.

## REFERENCES

- Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283.
- Allamanis, M., Barr, E. T., Devanbu, P., and Sutton, C. (2018). A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37.
- Black, P. E. (2018). *Juliet 1.3 Test Suite: Changes From 1.2*. US Department of Commerce, National Institute of Standards and Technology.
- Bojanowski, P., Grave, E., Joulin, A., and Mikolov, T. (2017). Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146.
- Chollet, F. et al. (2015). <https://github.com/fchollet/keras>.
- CVE (2019). Common vulnerabilities and exposures website. <https://cve.mitre.org/>.
- Fang, Y., Liu, Y., Huang, C., and Liu, L. (2020). Fastembed: Predicting vulnerability exploitation possibility based on ensemble machine learning algorithm. *Plos one*, 15(2):e0228439.
- 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., et al. (2018). Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497*.
- Henkel, J., Lahiri, S. K., Liblit, B., and Reps, T. (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*, pages 163–174.
- Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.
- Joulin, A., Grave, E., Bojanowski, P., and Mikolov, T. (2016). Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*.
- Kim, Y. (2014). Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*.
- Kostadinov, S. (2017). Understanding gru networks. <https://www.towardsdatascience.com>. Accessed 25 Jan 2020.
- Kovalenko, V., Bogomolov, E., Bryksin, T., and Bacchelli, A. (2019). Pathminer: a library for mining of path-based representations of code. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 13–17. IEEE Press.
- Kula, M. (2019). A python implementation of glove: glove-python. <https://github.com/maciejkula/glove-python>.

- Li, Z., Zou, D., Tang, J., Zhang, Z., Sun, M., and Jin, H. (2019). A comparative study of deep learning-based vulnerability detection system. *IEEE Access*, 7:103184–103197.
- Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., and Zhong, Y. (2018). Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*.
- Lin, G., Xiao, W., Zhang, J., and Xiang, Y. (2019a). Deep learning-based vulnerable function detection: A benchmark. In *International Conference on Information and Communications Security*, pages 219–232. Springer.
- Lin, G., Zhang, J., Luo, W., Pan, L., De Vel, O., Montague, P., and Xiang, Y. (2019b). Software vulnerability discovery via learning multi-domain knowledge bases. *IEEE Transactions on Dependable and Secure Computing*.
- Manning, C. D., Raghavan, P., and Schütze, H. (2009). *Introduction to Information Retrieval*. Cambridge University Press.
- Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.
- Mokhov, S. A., Paquet, J., and Debbabi, M. (2014). The use of nlp techniques in static code analysis to detect weaknesses and vulnerabilities. In *Canadian Conference on Artificial Intelligence*, pages 326–332. Springer.
- Niu, W., Zhang, X., Du, X., Zhao, L., Cao, R., and Guizani, M. (2020). A deep learning based static taint analysis approach for iot software vulnerability location. *Measurement*, 152:107139.
- NSCLab (2020). Cyber code intelligence github website. <https://github.com/cybercodeintelligence/CyberCI>.
- NVD (2019). National vulnerability database website. <https://nvd.nist.gov/>.
- Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543.
- Pradel, M. and Sen, K. (2017). Deep learning to find bugs. *TU Darmstadt, Department of Computer Science*.
- Ram, A., Xin, J., Nagappan, M., Yu, Y., Lozoya, R. C., Sabetta, A., and Lin, J. (2019). Exploiting token and path-based representations of code for identifying security-relevant commits. *arXiv preprint arXiv:1911.07620*.
- Řehůřek, R. and Sojka, P. (2010). Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta. ELRA.
- Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., and McConley, M. (2018). Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 757–762. IEEE.
- SARD (2019). Software assurance reference dataset project. <https://samate.nist.gov/SRD/>.