An Empirical Evaluation of the Usefulness of Word Embedding Techniques in Deep Learning-based Vulnerability Prediction

Ilias Kalouptsoglou[1, 2], Miltiadis Siavvas[1], Dionysios Kehagias[1], Alexandros Chatzigeorgiou[2], and Apostolos Ampatzoglou[2]

[1]Centre for Research and Technology Hellas, Thessaloniki, Greece

[2]University of Macedonia, Thessaloniki, Greece

{iliaskaloup@iti.gr, siavvasm@iti.gr, diok@iti.gr, achat@uom.edu.gr, a.ampatzoglou@uom.edu.gr }

Abstract

Software security is a critical consideration for software development companies that want to provide their customers with high-quality and dependable software. The automated detection of software vulnerabilities is a critical aspect in software security. Vulnerability prediction is a mechanism that enables the detection and mitigation of software vulnerabilities early enough in the development cycle. Recently the scientific community has dedicated a lot of effort on the design of Deep learning models based on text mining techniques. Initially, Bag-of-Words was the most promising method but recently more complex models have been proposed focusing on the sequences of instructions in the source code. Recent research endeavors have started utilizing word embedding vectors, which are widely used in text classification tasks like semantic analysis, for representing the words (i.e., code instructions) in vector format. These vectors could be trained either jointly with the other layers of the neural network, or they can be pre-trained using popular algorithms like word2vec and fast-text. In this paper, we empirically examine whether the utilization of word embedding vectors that are pre-trained separately from the vulnerability predictor could lead to more accurate vulnerability prediction models. For the purposes of the present study, a popular vulnerability dataset maintained by NIST was utilized. The results of the analysis suggest that pre-training the embedding vectors separately from the neural network leads to better vulnerability predictors with respect to their effectiveness and performance.

Introduction

Vulnerability Prediction (VP) techniques aim to identify the software components that are more likely to contain vulnerabilities. Vulnerability prediction models (VPMs) are typically built using machine learning (ML) techniques that use software attributes as input to differentiate between vulnerable and clean (or neutral) software components. Several VPMs have been proposed over the years, each of which uses different software factors as inputs to predict the presence of vulnerable components [1]. Text mining-based techniques have been found to be the most reliable, according to the bibliography [2] and have attracted the most of the recent research interest [3], [4], [5], [6], [7], [8], [9], [10]. The first attempts in the field of vulnerability prediction using text mining, have focused on the concept of Bag-of-Words (BoW) as a method for predicting software vulnerabilities using the text terms and their respective appearance frequencies in the source code. Recently, researchers have shifted their focus from simple BoW to more complex approaches, investigating whether more complex textual patterns in the source code could lead

to more accurate vulnerability prediction. In particular, the authors in [4], [5], [8] transformed the source code into sequences of word tokens and trained deep neural networks capable of learning sequences of data (e.g., Recurrent Neural Networks).

When using sequences of tokens to identify software components with vulnerabilities, vulnerability prediction has a lot in common with text classification tasks such as sentiment analysis [11]. The word embedding vectors are commonly used in the field of text classification. Word embedding is a term used to describe the representation of words for text analysis, typically in the form of a real-valued vector that encodes the meaning of the word in such a way that words that are close in the vector space are expected to have similar meanings. Most of the studies about VP make use of the word embeddings [4], [5], [6], [12]. Actually, without these embedding vectors each token will have been replaced by a one-hot vector with dimension equal to the size of vocabulary, making the whole process very time and memory consuming.

Tokens can be embedded into vectors in a variety of ways. One option is to use an embedding layer that is jointly trained with the vulnerability prediction task [13]. Another method is to use an external word embedding tool, such as word2vec [14], to generate vector representations of each token. One can also use the vectors that are already generated from these tools (e.g. word2vec, Glove [15], Fast-text [16]) based on natural language documents of billions of words. Finally, there is also the option to produce custom embedding representations.

The purpose of this study is to emerge the worth of the sophisticated embedding algorithms (e.g., word2vec, fast-text) in text mining-based vulnerability prediction showing their contribution to the effectiveness and the efficiency of the VPMs and to compare them with the use of a trainable embedding layer that updates its values during the training of the VP classifier. A dataset has been collected and an experimental analysis has been conducted by comparing the use of a simple embedding layer with the utilization of word2vec and fast-text algorithms. We also compare the word2vec and fast-text algorithms with each other. Finally, we compare our best model with a state-of-the-art model based on BoW and ML. All the produced results are presented in this paper.

The remainder of the paper is organized as follows. Section 2 discusses related work regarding the utilization of word embeddings in the field vulnerability prediction. Section 3 provides the theoretical background in order to familiarize the reader with the main concepts of the present work. Section 4 discusses the methodology that we followed, while Section 5 presents the results of our analysis. Finally, Section 6 wraps up the paper and discusses future research directions.

Related Work

Vulnerability prediction using text mining is very popular and has demonstrated promising results in the related literature [2], [4], [9], [10]. Initial research endeavors focused on the concept of BoW (i.e., occurrences of tokens) [2][9]. Recent attempts focus on predicting the existence of vulnerabilities through learning more complex patterns from the source code. They consider the software components as sequences of tokens and train deep learning models capable of learning sequences, such as the Recurrent Neural Networks (RNNs) [4], [5]. The challenging part of these recent studies is to add syntactic and semantic meaning to the sequences of code tokens. Word embeddings are one of the most promising solutions.

The word embedding vectors have evolved into an integral part of the text classification tasks since Mikolov et al. [17] proposed two architectures for learning distributed representations of words. The authors in [18], conducted a comparative study between different ML algorithms including fast-text, Glove and word2vec, while in [19] a deep learning method is proposed utilizing the semantic knowledge provided by the word embeddings.

Word embeddings have already been used in the field of text mining-based vulnerability prediction. Dam et al. [8] mapped every code token with an index of their vocabulary and then they constructed an embedding matrix which contained a unique vector representation for every token of the vocabulary in the position that corresponds to the vocabulary index. In other words, the embedding matrix worked as a look-up table.

The authors in [5] and [6] used the word2vec tool to generate embedding vectors for their vocabulary, while Zhou et al. [4] used the pre-trained word2vec vectors. Russel et al. in [12] created a vulnerability detection tool based on deep learning and capable of interpreting lexical source code. They conducted a comparative study between simple source code embedding using Bag-of-Words and more advanced code representations learned automatically by deep learning models inside the embedding layer. Fang et al. [20] proposed the fastEmbed model which is an extension of the fast-text algorithm. This way they developed a model for predicting the exploitability of software vulnerabilities on imbalanced datasets by understanding key features of vulnerability-related text.

To this end, it is quite clear that a lot of studies make use of word embedding vectors as a representative format for the source code's tokens (i.e., words). There are papers that refer the use of simple vector representations just in order to replace the text features [8], other papers that use the BoW methodology to represent the text in the source code [9], other studies that utilize the pre-trained embedding vectors produced by the pre-trained word2vec model [4], but most of them choose to encode the code tokens into embedding vectors trained on their own data [5], [6]. However, to the best of our knowledge, there is no study examining the difference between the internal embeddings that are trained in the embedding layer together with the classifier, and the external embeddings that are trained alone prior to the model's training. The former are part of the supervised learning of the model and update their weights through the Backpropagation process [21], while the latter are trained once, using an advanced unsupervised algorithm, and then they can be saved for future use. Moreover, there is a need for an experimental analysis examining the improvement in terms of accuracy and performance that these word embeddings provide to the DL-based vulnerability predictors. In the present work, we attempt to address these open issues through an empirical analysis on a popular dataset. Furthermore, the present paper includes a comparison between two popular types of word embedding tools (i.e., word2vec, fast-text) as well as a comparison with a state-of-the-art BoW model.

Theoretical Background

In this section, we present the theoretical background of the technologies that we use. This section's information is critical for familiarizing the reader with the concepts of the text mining-based VP and the word embedding representations.

Vulnerability Prediction based on Text-Mining

Vulnerability Prediction purpose is to identify software hotspots that are more likely to contain software vulnerabilities. These hotspots are actually parts of the source code that require more attention by the software developers and engineers from a security viewpoint. When the VPMs are based on text-mining they are trained on datasets constructed by the words (i.e., tokens) that appear in the source code. BoW constitutes the simplest text-mining method. In BoW, the code is divided into text tokens, each one of which is accompanied by the number of its occurrences in the source code. So each word corresponds to a feature, and the frequency of that feature in a component adds up to the value of that feature for that component. Aside from BoW, text-mining includes the process of converting the source code into a list of token sequences for use as input to Deep Learning (DL) models capable of parsing sequential data (e.g., recurrent neural networks). The sequences of tokens constitute the input of the DL models that, during the training phase, try to capture the syntactic information included in the source code, and in the execution phase to predict the existence of vulnerabilities in the software components. Text-mining also uses Natural Language Processing (NLP) methodologies such as word2vec pre-trained embedding vectors to extract semantic information from tokens.

Word Embedding Vectors

Word embedding methods use a corpus of text to learn a real-valued vector representation for a predefined fixed-sized vocabulary [17]. The learning process is either collaborative with the neural network model on a task, or unsupervised, using document statistics. An embedding layer is a word embedding learned in conjunction with a neural network model on a specific natural language processing task, such as document classification. It necessitates cleaning and preparing the document text so that each word can be one-hot encoded. The model specifies the size of the vector space. The vectors are seeded with small random numbers. The embedding layer is used at the front end of a neural network and is fitted in a supervised manner using the Backpropagation algorithm. However, it can be selected to be non-trainable. In this case, it has to be seeded with a pre-trained embedding matrix which has been trained using an external algorithm.

Mikolov et al. [17] proposed two model architectures for computing continuous vector representations of words. They showed that these representations were able to capture syntactic and semantic word similarities. Both architectures are neural network-based ones for learning the underlying word representations for every word. The first proposed model, called Continuous Bag-of-Words Model (CBOW), tends to find the probability of a word occurring in a context. Thus, it generalizes over all the different contexts in which a word can be used. The second architecture, called continuous skip-gram model, instead of predicting the current word based on context, attempts to maximize classification of a word based on another word in the same sentence. To be more specific, every current word is fed into a log-linear classifier with a continuous projection layer, which predicts words within a certain range before and after the current word.

Two of the most popular algorithms that can generate embedding vectors are the word2vec[1] and fast-text[2] models. Both of them are based on the two aforementioned architectures (i.e., CBOW,

[1] https://radimrehurek.com/gensim/models/word2vec.html
[2] https://radimrehurek.com/gensim/models/fasttext.html

skip-gram). The difference between these tools lies in the fact that the word2vec considers each individual word to be the smallest unit for which a vector representation must be found, whereas fast-text considers a word to be formed by n-grams of character and therefore fast-text is generally better in finding the vector representation for rare words.

Methodology

Dataset

As part of the current work, we created several VPMs for two widely-used programming languages, C and C++ combined. We used a vulnerability dataset derived from two National Institute of Standards and Technology (NIST) data sources: the National Vulnerability Database (NVD)[3] and the Software Assurance Reference Dataset (SARD)[4] . This dataset contains 7651 class files, 3438 of which are classified as vulnerable and the remaining 4213 as clean. The dataset has been presented by Li et al. [5].

Pre-Processing

Before the construction of vulnerability prediction models, appropriate pre-processing is required in order to bring the dataset in a form appropriate to be used by the investigated techniques. To this end, we gathered the source code files written in the C and C++ programming languages and used a variety of pre-processing techniques to convert the datasets into a series of words-tokens. All comments, as well as the header/import instructions that declare the use of specific libraries in the class, were removed from the dataset. Subsequently, we removed the code-specific constants (i.e., numbers, literals, etc.), in order to make the produced sequences more generalizable. In particular, the numeric values (i.e., integers, floats, etc.) were then replaced by a unique identifier "numId$," while the string values and characters were replaced by a different unique identifier "strId$." All blank lines are also removed, and the text is finally transformed into a list of code tokens (i.e., new, char, strlen, etc.) in the order they appear in the source code. After data cleansing, these produced tokens are replaced by a unique integer (integer encoding process[5]) and these integers are mapped to one-hot vectors (one-hot encoding[6]). The aforementioned data cleansing process is illustrated in Figure 1.

[3] https://nvd.nist.gov/
[4] https://samate.nist.gov/SRD/index.php
[5] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html
[6] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html

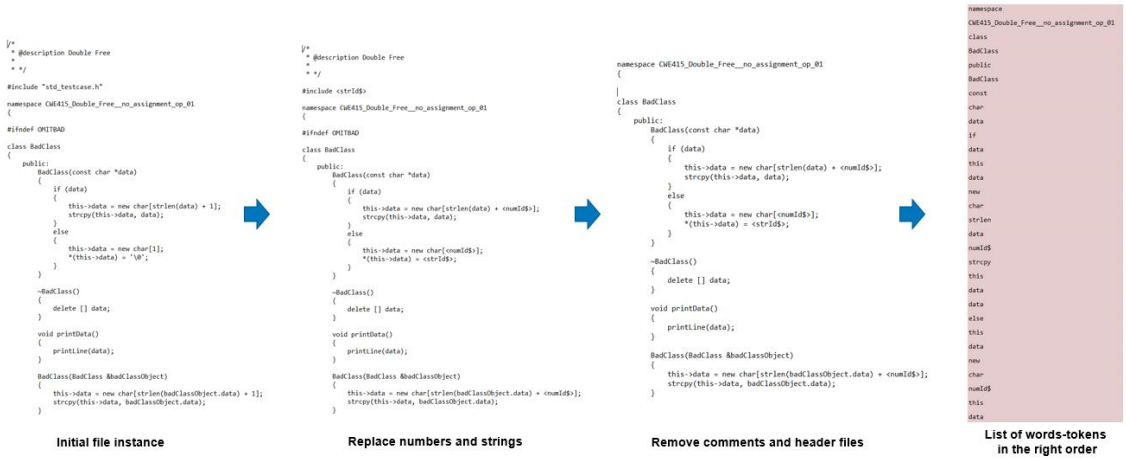

Figure 1: The data cleansing and tokenization process.

Word Embedding Vectors Training

In this study, in order to embed the text representations to numerical vectors different from the one-hot vectors, the word2vec and fast-text tools were utilized. The two models provide both the CBOW and skip-gram architecture. These tools also provide pre-trained vectors for a specific but large vocabulary. However, in our case, with a total vocabulary size equal to 11992 programming language words, the pre-trained (in natural language text) representations would not provide any improvement. Instead of that, we train the tools with our dataset. Each software component (i.e., source code file) constitutes a sequence of tokens and all the sequences of the dataset are used as the corpus for the training of the word2vec and fast-test models. These algorithms learn the syntactic and semantic relations between the code tokens and place them at the vector space. After training these embedding vectors for the words of the vocabulary then one can save them for future use, saving time of the training process. A comparative study between the CBOW and skip-gram architectures and between the word2vec and fast-text vectors is conducted in the Section 5. For the training of the embedding vectors, the parameters that were selected after tuning are listed in Table 1.

Table 1: The selected parameters for the training of word2vec and fast-text embedding vectors.

Parameters	**Word2vec**	**Fast-text**
size	300	300
window	40	40
min_count	1	1
Epochs	1	2

In Table 1, the parameter “size” is the dimension of the embedding vectors, the “window” refers to the maximum distance between a target word and words surrounding the target word while the term “min_count” refers to the minimum count of words to consider during training. The algorithm ignores the words with occurrence less than the “min_count”. The parameter epochs is just the number of iterations that the model parses the data.

In Figure 2, there is a depiction of word2vec vectors trained at the dataset used in the present study, generated by the t-distributed stochastic neighbor embedding (TSNE) algorithm [22]. Vectors that are in close proximity in the depicted figure, correspond to words that are in close proximity in the actual source code. For instance, in Figure 2 we can see that the tokens “for” and “i” which actually are used together in a lot of circumstances, are indeed placed one next to the other. The same applies also for the tokens “free” and “malloc”, which is another very representative example.

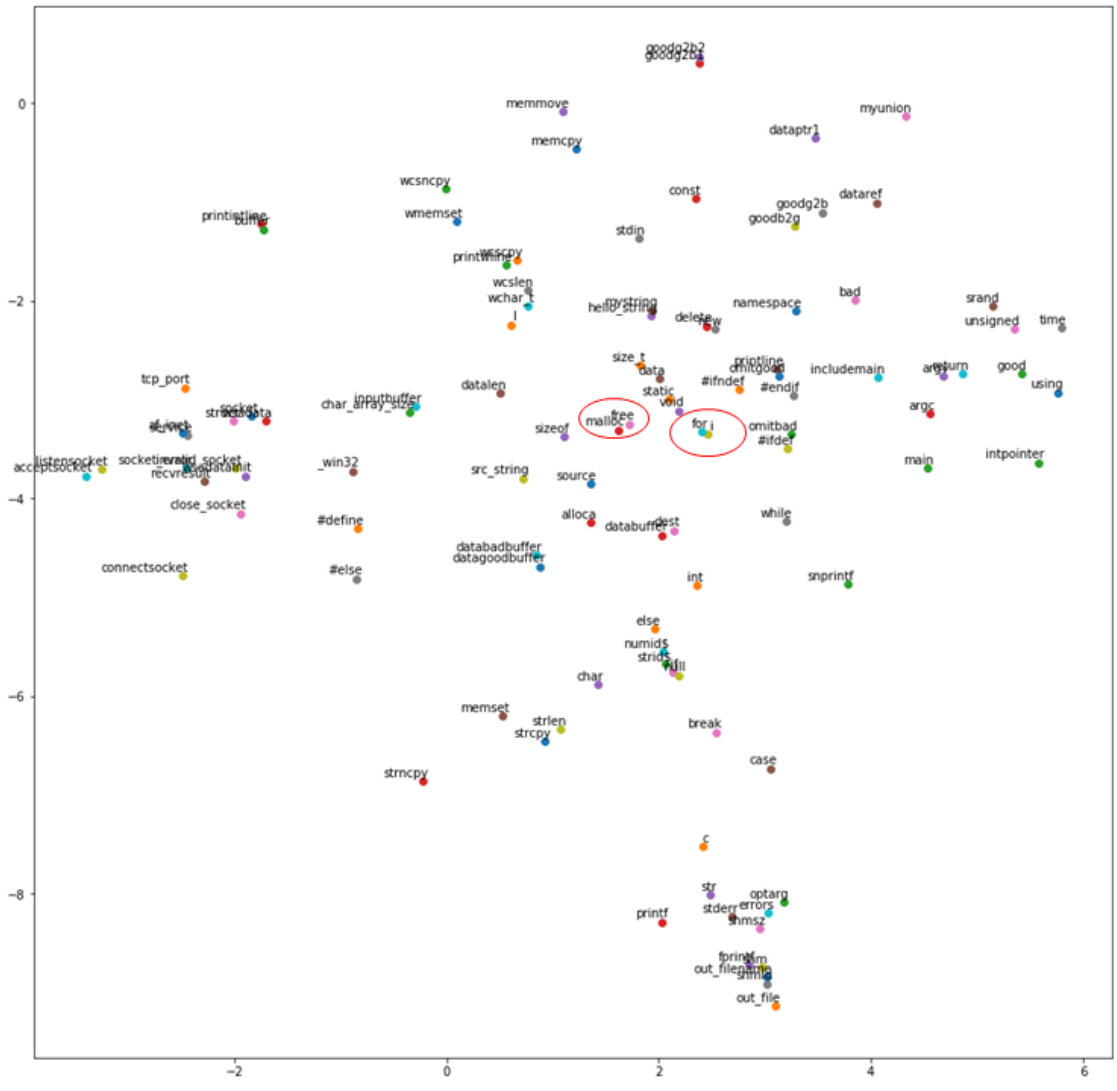

Figure 2: The word2vec embedding vectors placed in the vector space by the TSNE algorithm.

Model Selection

In this analysis, various DL algorithms are used to create models that can distinguish between vulnerable and neutral source code files. As the input to the models consists of sequential data (i.e., series of tokens) we chose DL algorithms capable of handling sequences. The RNNs are the most suitable ones as language models [23]. Convolutional Neural Networks (CNNs) are used on code classification tasks, as well [4], [24]. Regarding the RNNs, there are several improved versions

such as the Long-Short Term Memory networks (LSTMs) [25], the Gate Recurrent Units (GRUs) [26] and the Bidirectional LSTMs (BiLSTMs) [27], which can solve the vanishing gradient problem [28] that the original RNNs face. The hyper-parameters chosen for our RNN and CNN models are presented in Table 2 and Table 3 respectively. Their values were selected after consecutive tuning and re-evaluation.

Table 2: The selected Hyper-parameters of the RNNs.

Hyper-parameter Name	**Value**
Number of Layers	3 (Embedding-Recurrent-Dense)
Number of Recurrent Layers	1 (LSTM/GRU/BiLSTM)
Embedding Size	300
Number of Hidden Units	300
Weight Initialization Technique	Glorot Uniform (Xavier)
Learning Rate	0.01
Gradient Descent Optimizer	Adam
Batch Size	64
Activation Function	relu
Output Activation Function	sigmoid
Loss Function	Binary cross entropy
Over-fitting Prevention	Dropout = 0.3
Maximum Epochs	100
Early Stopping Patience	10

Regarding the CNN model that we trained, the selected Hyper-parameters are the following:

Table 3: The selected Hyper-parameters of the CNN.

Hyper-parameter Name	**Value**
Number of Layers	3 (Embedding- Convolutional -Dense)
Number of Convolutional Layers	1 (1D CNN)
Embedding Size	300
Number of Filters	128
Kernel Size	5
Pooling	Global Max Pooling
Weight Initialization Technique	Glorot Uniform (Xavier)
Learning Rate	0.01
Gradient Descent Optimizer	Adam
Batch Size	64
Activation Function	relu
Output Activation Function	sigmoid
Loss Function	Binary cross entropy
Maximum Epochs	100
Early Stopping Patience	10

Evaluation Metrics

Several evaluation metrics are available in the literature and are commonly used to assess the predictive effectiveness of the ML models. The number of True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN) produced by the models is typically used to calculate these performance indicators. In the vulnerability prediction case, a special emphasis is placed on the Recall (R) of the produced models, because the higher the Recall of the model is, the more real vulnerabilities it predicts. Apart from the capability of the produced models to identify the great majority of vulnerable files contained in a software project, the volume of the produced FP (i.e., clean files marked as vulnerable by the models) is important to consider because it is known to affect the models' utilization in practice. If the number of FP is large, developers will have to inspect an important number of non-vulnerable files in order to detect a vulnerable file. As a result, the number of FP is closely related to the amount of manual effort required by developers to identify files that contain actual vulnerabilities. The lower the number of FP is, the higher the precision of the model. So we have to consider both recall and precision. This fact emphasizes the significance of the f1-score, which represents the balance of precision and recall. However, because identifying vulnerable files at the expense of producing FP is more important in VP, we chose f2-score as our evaluation metric in order to tune our models and evaluate them in the testing dataset. The f2-score is a weighted average of precision and recall, with recall being more important than precision. It is equal to:

$$F_2 = 5\frac{precision \times recall}{4 \times preciion + recall}$$

Results and Discussion

In this section, we present the results of our analysis and discuss the outcome of the experiments. All the experiments were conducted in NVIDIA GeForce GTX 1660 using the CUDA[7] platform. The training of the DL model was performed using the tensorflow keras library. Table 4 reports the evaluation results of the DL models that were built based on the sequences of tokens in the source code. This table sums up the results regarding the f2-score for all the RNN variations and CNN using word2vec or fast-text embeddings in contrast with the joint training of the embeddings with the neural network's training. Ten-fold cross-validation process was employed. The dataset is divided into 10 folds in 10-fold cross-validation, with 9 participating in training and the remaining one participating in evaluation. Every time, the fold that remains for evaluation is different. As a result, we have a model that has been trained and evaluated 10 times using different training and testing data each time. The model's performance is the average of these 10 models' performances. As a result, the possibility of biased results is eliminated.

Table 4: The f2-score of all the utilized methods after 10 fold cross-validation.

	Simple Embedding Layer	**Word2vec CBOW**	**Word2vec Skip-gram**	**Fast-Text CBOW**	**Fast-Text Skip-gram**
LSTM	77.98	85.11	88.38	84.92	88.66
BiLSTM	80.28	85.86	88.01	82.33	86.04

[7] https://developer.nvidia.com/cuda-toolkit

GRU	72.22	89.15	87.94	84.28	89.10
CNN	81.46	86.36	89.43	86.36	84.54
Average	77.99	86.62	88.44	84.47	87.09

From Table 4, we can see that the use of sophisticated word embeddings trained prior to the deep learning model is beneficial at each model case. The average f2-score of the four models when using an algorithm for the generation of the word embedding vector is significantly bigger. Furthermore, it is clear that the skip-gram model is better than the CBOW in our dataset as it achieves greater average f2-score both at the word2vec and the fast-text case. Similarly, we notice that the word2vec method provides better f2-score, both at the CBOW and the skip-gram variation, compared with the fast-text embeddings. All the aforementioned foundings lead to the conclusion that the skip-gram variation of the word2vec embeddings is the best choice for embedding the tokens of the source code of our dataset before giving them as input to the sequential deep learning model. An 11% increase in terms of f2-score when using word2vec embeddings compared with the trainable embedding layer is a significant improvement and indicates to the initial hypothesis that these sophisticated models are capable of capturing semantic and syntactic relationships between the words of the source code.

Furthermore, the training of the embedding vectors outside from the embedding layer (i.e., non-trainable embedding layer) is beneficial not only in accuracy but also in terms of performance. The training time of the DL models has decreased significantly. Table 5 sums up the results about the training time.

Table 5: The training time in milliseconds (ms) both in case of trainable embedding layer and in case of sophisticated embeddings trained independently of the neural network.

	Simple Embedding Layer	**Word2vec-Skip-gram**
LSTM	13078 ms	9090 ms
BiLSTM	22596 ms	18011 ms
GRU	12025 ms	8330 ms
CNN	9774 ms	4276 ms

From Table 5, it is clear that the training times when having ready the embedding vectors are by far smaller compared with the case of joint training along with the rest layers. Another interesting note derived from Table 4 and 5 is the fact that the CNN model is more accurate than the RNNs and much faster as well.

Finally, another interesting question would be to examine whether the adoption of word embedding vectors lead to better vulnerability prediction models compared to the traditional (and simpler) BoW approach. For this purpose, we compare our best model that utilizes the word embedding concept to the best model that uses BoW and is trained and evaluated on the same dataset. In particular, in Table 6, we present the results of the comparison between the state-of-

the-art BoW method, versus the CNN model with skip-gram word2vec vectors, which was found to be the best model in our previous analysis. In the case of BoW, we chose Random Forest [29] (composed of 100 trees) as a classifier, based on bibliography [2], [9], [30]. From Table 6, it is observed that the f2-score is greater in the case of using sophisticated embeddings. Actually, these word2vec vectors can be used only at token series models (i.e., CNN, RNN) and not in BoW, constituting a major drawback of the method.

Table 6: BoW versus CNN that uses the skip-gram word2vec representations.

	Accuracy	**Precision**	**Recall**	**F2-score**
BoW	88.69	90.40	85.80	86.66
Skip-gram word2vec with CNN	88.25	86.21	90.31	89.43

Conclusion and Future Work

In this paper, we investigated the usefulness of the numerical representations of the source code words, with the aim of predicting vulnerabilities. We focused on examining whether the utilization of sophisticated (i.e., external) embedding vectors is beneficial in contrast with the training of the embedding vectors jointly with the vulnerability predictor. Moreover, a comparison between the CBOW and the continuous skip-gram architectures took place as well as a comparison between the word2vec and fast-text algorithms. We used a C/C++ dataset provided by NVD and SARD for training and evaluating our models.

We showed that either the word2vec or fast-text methodologies provide better results than the trainable embedding layer which is trained along with the rest layers of the neural network. These vector representations seem able to capture semantic and syntactic relations between the words in the code and so they can be proved beneficial when training models on sequences of code tokens. The word2vec method proved to be superior to fast-text when applied in our dataset. Furthermore, the skip-gram model demonstrated better scores compared with the CBOW, both in cases of word2vec and fast-text. Another important advantage of these sophisticated vectors is the time reduction during the model training, as there is no need to train the embedding layer again. Last but not least, the CNN with trained word2vec embeddings, which appeared to be our best model, demonstrates higher f2-score than the BoW model.

There are several potential directions for future work. First of all, the present study was based on a dataset containing exclusively C/C++ code. We intend to replicate our study using software products written in other programming languages (e.g., Java, Python, etc.) to investigate the generalizability of the produced results. Furthermore, another interesting topic would be to examine whether the process of embedding the source code in a higher level of granularity (e.g., line or function level) could be proved beneficial for vulnerability prediction.

References

[1] M. Siavvas, E. Gelenbe, D. Kehagias, and D. Tzovaras, "Static analysis-based approaches for secure software development," in *International ISCIS Security Workshop*, 2018, pp. 142–157.

[2] "Predicting vulnerable components: Software metrics vs text mining," in *2014 IEEE 25th international symposium on software reliability engineering*, 2014, pp. 23–33.

[3] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *IEEE Trans. Softw. Eng.*, 2021.

[4] "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *arXiv Prepr. arXiv1909.03496*, 2019.

[5] "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv Prepr. arXiv1801.01681*, 2018.

[6] "BGNN4VD: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection," *Inf. Softw. Technol.*, vol. 136, p. 106576, 2021.

[7] "Predicting vulnerable software components through deep neural network," in *Proceedings of the 2017 International Conference on Deep Learning Technologies*, 2017, pp. 6–10.

[8] "Automatic feature learning for predicting vulnerable software components," *IEEE Trans. Softw. Eng.*, 2018.

[9] "Predicting vulnerable software components via text mining," *IEEE Trans. Softw. Eng.*, vol. 40, no. 10, pp. 993–1006, 2014.

[10] "Software vulnerability prediction using text analysis techniques," in *Proceedings of the 4th international workshop on Security measurements and metrics*, 2012, pp. 7–10.

[11] "Sentiment analysis algorithms and applications: A survey," *Ain Shams Eng. J.*, vol. 5, no. 4, pp. 1093–1113, 2014.

[12] "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, 2018, pp. 757–762.

[13] J. Turian, L. Ratinov, and Y. Bengio, "Word representations: a simple and general method for semi-supervised learning," in *Proceedings of the 48th annual meeting of the association for computational linguistics*, 2010, pp. 384–394.

[14] X. Rong, "word2vec parameter learning explained," *arXiv Prepr. arXiv1411.2738*, 2014.

[15] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[16] "Bag of tricks for efficient text classification," *arXiv Prepr. arXiv1607.01759*, 2016.

[17] "Efficient estimation of word representations in vector space," *arXiv Prepr.*

arXiv1301.3781, 2013.

[18] R. A. Stein, P. A. Jaques, and J. F. Valiati, “An analysis of hierarchical text classification using word embeddings,” *Inf. Sci. (Ny).*, vol. 471, pp. 216–232, 2019.

[19] Y. Ma, H. Peng, and E. Cambria, “Targeted aspect-based sentiment analysis via embedding commonsense knowledge into an attentive LSTM,” 2018.

[20] Y. Fang, Y. Liu, C. Huang, and L. Liu, “FastEmbed: Predicting vulnerability exploitation possibility based on ensemble machine learning algorithm,” *PLoS One*, vol. 15, no. 2, p. e0228439, 2020.

[21] “Neural network methods for natural language processing,” *Synth. Lect. Hum. Lang. Technol.*, vol. 10, no. 1, pp. 1–309, 2017.

[22] “Visualizing data using t-SNE.,” *J. Mach. Learn. Res.*, vol. 9, no. 11, 2008.

[23] “Comparison of feedforward and recurrent neural network language models,” in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013, pp. 8430–8434.

[24] K. Filus, M. Siavvas, J. Domańska, and E. Gelenbe, “The random neural network as a bonding model for software vulnerability prediction,” in *Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2020, pp. 102–116.

[25] “LSTM neural networks for language modeling,” 2012.

[26] “Empirical evaluation of gated recurrent neural networks on sequence modeling,” *arXiv Prepr. arXiv1412.3555*, 2014.

[27] “Bidirectional recurrent neural networks,” *IEEE Trans. Signal Process.*, vol. 45, no. 11, pp. 2673–2681, 1997.

[28] “The vanishing gradient problem during learning recurrent neural nets and problem solutions,” *Int. J. Uncertainty, Fuzziness Knowledge-Based Syst.*, vol. 6, no. 02, pp. 107–116, 1998.

[29] L. Breiman, “Random forests,” *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.

[30] “Combining software metrics and text features for vulnerable file prediction,” in *2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2015, pp. 40–49.

Appendix

In the Appendix Section we provide some extra Tables of the results produced by our study, including values for accuracy, precision and recall aside from the f2-score.

Simple Embedding Layer:

	Accuracy	**Precision**	**Recall**	**F2-score**
LSTM	75.61	74.69	79.60	77.98
BiLSTM	77.05	76.06	82.73	80.28
GRU	73.06	72.60	72.34	72.22
CNN	85.15	88.43	79.96	81.46

Word2vec embeddings - CBOW:

	Accuracy	**Precision**	**Recall**	**F2-score**
LSTM	78.76	74.23	88.90	85.11
BiLSTM	80.62	75.79	88.99	58.86
GRU	84.55	79.52	92.02	89.15
CNN	86.60	86.16	86.52	86.39

Word2vec embeddings – skip-gram:

	Accuracy	**Precision**	**Recall**	**F2-score**
LSTM	84.07	79.35	91.14	88.38
BiLSTM	84.51	80.26	90.31	88.01
GRU	83.44	78.51	90.74	87.94
CNN	88.25	86.21	90.31	89.43

Fast-text embeddings - CBOW:

	Accuracy	**Precision**	**Recall**	**F2-score**
LSTM	79.79	75.11	88.07	84.92
BiLSTM	76.20	71.15	85.79	82.33
GRU	77.58	72.71	88.53	84.28
CNN	86.08	84.98	86.82	86.36

Fast-text embeddings – skip-gram:

	Accuracy	**Precision**	**Recall**	**F2-score**
LSTM	83.54	78.23	91.79	88.66
BiLSTM	79.20	73.49	90.01	86.04
GRU	82.38	76.57	93.11	89.10
CNN	85.15	84.87	84.58	84.57