Jordanian Journal of Computers and Information Technology (JJCIT), Vol. 07, No. 01, March 2021.

EFFICIENT DEEP FEATURES LEARNING FOR VULNERABILITY DETECTION USING CHARACTER N-GRAM EMBEDDING

Mamdouh Alenezi[1], Mohammed Zagane[2] and Yasir Javed[1]

(Received: 19-Aug.-2020, Revised: 2-Oct.-2020 and 28-Oct.-2020, Accepted: 5-Nov.-2020)

ABSTRACT

Deep Learning (DL) techniques were successfully applied to solve challenging problems in the field of Natural Language Processing (NLP). Since source code and natural text share several similarities, it was possible to adopt text classification techniques, such as word embedding, to propose DL-based Automatic Vulnerabilities Prediction (AVP) approaches. Although the obtained results were interesting, they were not good enough compared to those obtained in NLP. In this paper, we propose an improved DL-based AVP approach based on the technique of character n-gram embedding. We evaluate the proposed approach for 4 types of vulnerabilities using a large c/c++ open-source codebase. The results show that our approach can yield a very excellent performance which outperforms the performances obtained by previous approaches.

KEYWORDS

Software security, Vulnerability detection, Deep features learning, Character N-gram embedding.

1. INTRODUCTION

Disastrous consequences related to exploiting software vulnerabilities can be avoided if these vulnerabilities are early detected and fixed before software deliverance. Many solutions to automatic vulnerabilities prediction (AVP) have been proposed. Manual vulnerable code detection is very hard and very costly, especially when dealing with software with a large codebase. These solutions aim to assist developers and minimize costs related to detection and fixing of vulnerabilities by letting them focus their effort and time on the components (files, classes or functions) that are most probable to be vulnerable. Researchers have proposed several approaches to develop vulnerability prediction models (VPMs) that are cable of discriminating vulnerable components from clean components. The most important works were to propose data-driven approaches based on using software attributes, such as software metrics with machine learning (ML) techniques to build VPMs. The major limitation of these approaches lies in the fact that important semantic and syntactic characteristics of the code that may give insight about vulnerabilities cannot be captured by using only static code attributes.

Motivated by the success of using deep learning (DL) techniques in other fields, such as natural language processing (NLP) and image processing, researchers in recent research works (see related work section) in the field of AVP begin to apply DL techniques to predict and locate vulnerabilities. Since source code shares several characteristics of the natural text and the same thing is valid for programming language and natural language (both have: vocabulary, syntactic and semantic characteristics, …etc), researchers have proposed to deal with source code written in a programming language like dealing with the natural text of a natural language. Therefore, techniques used in some applications of NLP, such as text classification, are adopted in the field of AVP to predict and locate vulnerabilities: classifying source code entities (file, function or slices) as vulnerable or clean (Figure 1). More specifically, the techniques, such as word embedding and bag-of-word used in NLP to automatically extract features from the natural text, are applied to automatically extract features from the source code. The automatically-extracted features are then used as input for a classifier based on machine learning (ML), which classifies source code as vulnerable or clean (Figure 1: solid lines). In DL-based approaches, the output of the first step of feature extraction (the input vectors) is passed to a deep neural network (DNN) to learn more hidden features (deep features). Since the important hidden

1. M. Alenezi and Y. Javed are with Department of Computer Science, Prince Sultan University, Riyadh, KSA. Emails: malenezi@psu.edu.sa, yjaved@psu.edu.sa
2. M. Zagane is with Department of Computer Science, University Mustapha Stambouli of Mascara, Mascara, Algeria. Email: mohamed.zaagane@univ-mascara.dz

features which become the actual classifier inputs are learned *via* the DNN, the first step of feature extraction (word embedding, bag-of-word, …etc) is considered in the DL-based approach as input vectorization.

Two main DL-based approaches are proposed (Figure 1: dashed lines). In the first approach, a DNN is used to deeply learn hidden features from the vectorized inputs and predict vulnerabilities (i.e., as a classifier), while in the second approach, a DNN is only used to learn hidden features which are then used as inputs (features) for an ML-based classifier that predicts vulnerabilities.

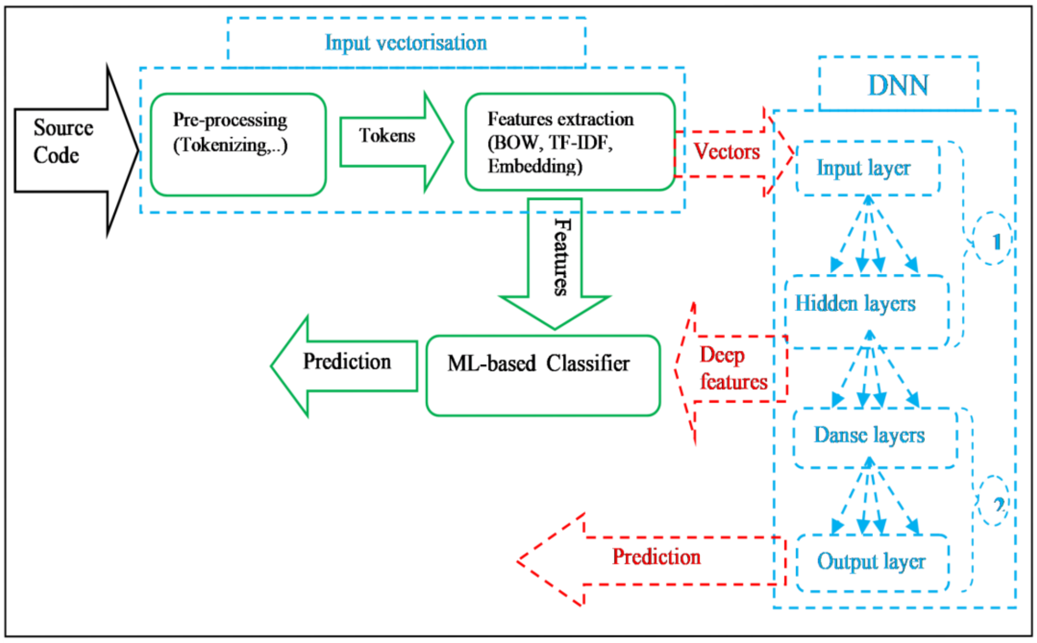

Figure 1. Vulnerabilities prediction approaches inspired by NLP techniques (dashed lines: DL-based approach, solid lines: ML-based approach) (1: learning deep hidden features, 2: classification).

Source code and natural text have main similarities that make it possible to adopt techniques used in NLP also in AVP. The most important adopted technique is the word embedding technique. The word or token embedding allows representing the words of a text in the form of vectors suitable to be processed by DNNs. The efficiency of this method compared to other methods, such as Bag-of-Word, is that it allows preserving semantic and syntactic information of words. On the other hand, there are characteristics which are specific to source code, making obtaining good performance very challenging. The most important characteristic is the large vocabulary and the rare words that can have source code. To address this problem, researchers proposed to apply vocabulary reduction methods. These methods allowed them to initiate using word embedding on the code, but at the expense of reduced performance. This represents a limitation, because these methods can cause a significant loss of valuable information related to vulnerabilities.

This research aims to address this limitation by proposing a code embedding solution that can be applied without reducing vocabulary, thus improving the vulnerability detection performance. The proposed approach lies in using N-gram-based embeddings at the character level. Compared with previous methods, the proposed embedding method can be applied without reducing vocabulary and enables the semantics of sub-tokens to be learned, which can avoid out-of-vocabulary tokens, thus reducing the possibility of information loss. Besides, we embed code at the slice granularity level, which allows the vulnerable code to be precisely identified.

The contribution of this work is two-fold:

- Proposing and evaluating an efficient and effective input vectorization approach based on the character n-gram embedding technique proposed by the Facebook research team [1]. The proposed approach allowed us to improve the performance of vulnerability detection.
- Proposing and making publically available a dataset generated following the proposed approach. This dataset can be used by other researchers in other research works or to train concrete vulnerability detection systems.

The remainder of this paper is organized as follows: in Section 2, we present the most relevant related works, while in Section 3, we describe the proposed approach and in Section 4, we present the experimental evaluation. In Section 5, we discuss the obtained results, while in Section 6, we highlight the limitations of the study and in Section 7, we summarize the work done in this study and indicate some perspectives for future works.

2. RELATED WORK

In this section, we present the most related works in the field of vulnerability prediction. To show the difference and the contribution of the recent DL-based approach, we begin by briefly presenting the previous ML/static-code-attributes-based approaches in the first sub-section, then in the second sub-section, we present DL/automatically-learned-features approaches. For the sake of brevity, we will focus only on the works that used the technique of word embedding to represent source code.

2.1 Traditional ML-based Approaches

Applying traditional ML techniques to predict software vulnerabilities has attracted the attention of several researchers. Indeed, considerable research works have been done to propose automatic vulnerability prediction (AVP) approaches based on machine learning (ML) and manually-defined static code features, such as software metrics ([2]–[9]) and text-based features [7], [10]. These works were motivated by the success of similar works [11]–[15] that have been done to predict software defects and by the fact that several code attributes, such as complexity, size and coupling (which can be quantified by corresponding software metrics), are proven in practice to be correlated to vulnerabilities. As reported in [16], the task of defining features is tedious, subjective and sometimes error-prone because of the complexity of the problem. This means that the quality of the resulting features and therefore the effectiveness of the resulting detection system varies with the individuals who define them. Another major drawback of these approaches lies in the fact that important semantic and syntactic characteristic of the code, which may give insight about vulnerabilities, cannot be captured by using only static code attributes. Another limitation of these approaches inherited by the coarse granularity level (file, class and method), in which software metrics are calculated, is that vulnerabilities cannot be located in much fine granularity. Recent works have tried to improve these approaches. Researchers in [9] have tried to combat the limitation of coarse granularity by proposing to calculate metrics at the slice granularity which allowed to improve the performance of the proposed VPMs (Precision: 95.1%, Recall: 95.0% FN Rate: 4.91%) and to locate with much precision the vulnerable lines. Other studies, such as [17]-[18], investigated using the automatically-learned features to build prediction models. However, the ML-based approaches still suffer from the missing semantic and syntactic features of the code and cannot learn deeply hidden features of the code which may exhibit a better way of characteristics of vulnerabilities. This is why in recent studies, researchers begin to use DL in AVP to benefit from the power of DL in learning hidden features. The most important of these studies are presented in the next sub-section.

2.2 DL-Based Approaches

DL techniques have been successfully applied to solve challenging problems in fields, such as NLP and image processing. Motivated by this success, researchers of the field of AVP in recent years begin investigating the application of DL techniques to predict and locate vulnerabilities in source code. The researchers' aim was essentially to benefit from the power of DNNs to learn deep hidden features that can perfectly characterize the vulnerable code, which was impossible using classic ML techniques, as well as to use them as a classifier (Figure 2).

Unlike the ML-based approach where several works have been carried out, few researchers have addressed AVP using DL techniques. The first use of DL in AVP was done by Catal et al. in [19]-[20]. In the first study, they conducted a literature review to investigate DL algorithms that can be applied in AVP. They concluded that, depending on the availability of the data, different kinds of DL algorithms can be applied in AVP: supervised learning models, unsupervised deep learning models or semi-supervised learning. In the second study, they proposed a web service-based VPM to predict vulnerable files of web applications. They used a dataset [21] proposed by [7] to train several machine learning techniques that exist in the Azure Machine Learning Studio environment and a Multi-Layer

Perceptron (MLP). They reported that the best performance (AUC: 76,5%) is achieved by the MLP. The type of VPMs' inputs was a set of code metrics. Researchers in [22] also used software metrics with DL to predict vulnerabilities. The authors investigated the usefulness of using software metrics as input for DNNs to locate vulnerabilities. Researchers reported that they used a large dataset [23] suitable for DL. The code metrics used as inputs for the DNNs (MLP and LSTM) were calculated at the slice ([24]–[26]) granularity level which allows them to locate the vulnerable lines of code. Based on comparing the obtained results (Recall: 73.9%, Precision: 74.4% and FN Rate: 26.14%) with the results reported by similar works that adopted techniques used in the field of NLP, the authors concluded that software metrics represent good -but not the best- data to use with DL-based approaches in AVP and that software metrics are more suitable for ML-based approaches which gave them very good results (Recall: 93.7%, Precision: 93.2% and FN Rate: 6.25%).

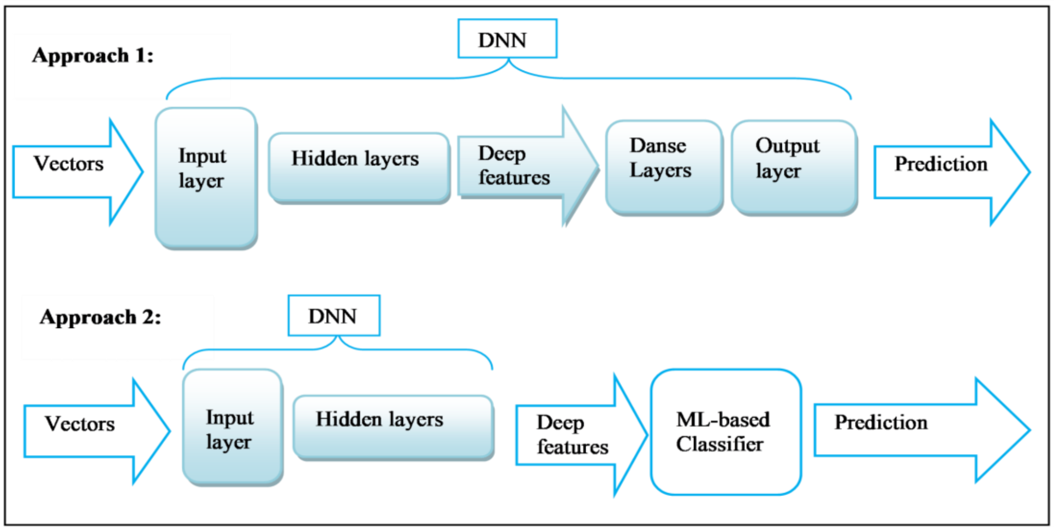

Figure 2. Using DNNs to predict vulnerabilities.

As we said before, the similarities between source code and the natural text have motivated researchers in the field of AVP to adopt techniques used in NLP to predict vulnerabilities. Essentially, techniques, such as word or token embedding used in NLP to "vectorize" inputs (representing text as vectors suitable to be used as inputs for DNNs), were adopted by recent works [16], [18], [27]–[30] to represent source code as vectors. The DNNs are used to learn from the vectorized inputs deep hidden features of the code that are related to vulnerabilities (Figure 1). Z. Li et al. in their works [16], [28]-[29] used the word2vec tool [31]-[32] which is based on using NNs to learn a vector representation of the word that preserves its semantic meaning based on its context, starting from the idea that words with similar meanings will tend to appear in contexts with similar words. Researchers in [18], [27] used custom embedding techniques inspired by previous works done on sentence classification, such as [33].

In all of these studies, token embedding was at the token level. This means that a distinct vector is assigned to each token in the training set. For an embedding model to be efficient and effective, it must provide representation for all or at least most of the words that compose the vocabularies. In AVP, this represents a challenging problem to solve, because source code may have very large vocabularies and many rare words induced by the fact that ways of writing code, especially the task of naming variables and functions, vary from developer to developer. To combat this problem, researchers used techniques to reduce vocabulary size by mapping user-defined variables and functions and all literal values (number and string) to special tokens. For example, in [18], all integers, real numbers, exponential notation and hexadecimal numbers are replaced with a generic <num> token and constant strings are replaced with a generic <str> token. Also, all rare tokens (e.g. occurring only once in the corpus) and tokens which exist in test sets but do not exist in the training set are replaced with a special token <unk>. In [16], [28]-[29], all user-defined variables and functions were mapped to representations, such as VAR1, VAR2, FUN1, FUN2, …etc.

Using these techniques, researchers were able to reduce the vocabularies size and partially benefit from the power of token embedding. However, these techniques of reducing vocabularies may lead to a very important loss of information by abstracting away certain syntactic and semantic characteristics of the code that are useful for vulnerability detection, which represents a limitation. To the best of our knowledge, no recent work has addressed this limitation. Instead, in recent works, researchers tried to address other aspects of DL-based AVP. In [30], researchers studied the cross-domain AVP. They proposed and evaluated a method to learn cross-domain representations in a range of cross-domain settings, including cross-project, cross-vulnerability and prediction of recent software vulnerabilities. Researchers in [34] addressed the problem of class imbalance between vulnerable code and non-vulnerable code. A new fuzzy oversampling method is proposed to rebalance the training data. In [35], both cross-project and class imbalance problems were studied.

To fill the research gap highlighted in the previous paragraph, we propose in this study an improved DL-based approach to detect vulnerabilities. Instead of reducing vocabularies and using token-level embedding, we propose an approach based on the works [1], [36] done by Facebook AI Research. The proposed approach is presented in detail in the next section.

3. Proposed Approach

The proposed approach is adopted from the communally used approach described in Figure1 (dashed lines) which was inspired by the previous works in the field of NLP. In our approach, DNNs are used to learn deep hidden features and as a classifier Figure 2 (approach 1). The different aspects of the proposed approaches: granularity level and input vectorization, are described in the following sub-sections.

3.1 Granularity Level

In previous studies of AVP, whether ML-based or DL-based, researchers investigated vulnerabilities prediction at different levels of granularity: file [7]-[8], [18], function/method [27], [37] and slice [9], [16], [22], [28]-[29]. Prediction at a coarse level (file and function) does not locate the vulnerable lines of code; instead, it can identify the components (files or functions) that require more focus from developers, which is less useful especially when the components are very large. Because the objective of the AVP is to assist developers and minimize the costs of vulnerabilities detection by minimizing the human intervention as much as possible, these coarse granularity levels are to be avoided.

A slice is a reduced version (few lines of code) of a source component automatically-extracted from the original component by analyzing its data flow and control flow in respecting a slicing criterion [25]. Slicing is useful in several software engineering applications, such as debugging, program comprehension and change impact prediction because it can give insight into multiple behavioral aspects of the source entity, such as all lines that change the value of a variable or that participate in computing the return value of a function [38]. In AVP, this can be useful, for example, to get all statements that are related to critical function calls (memory management, string manipulation, …etc.).

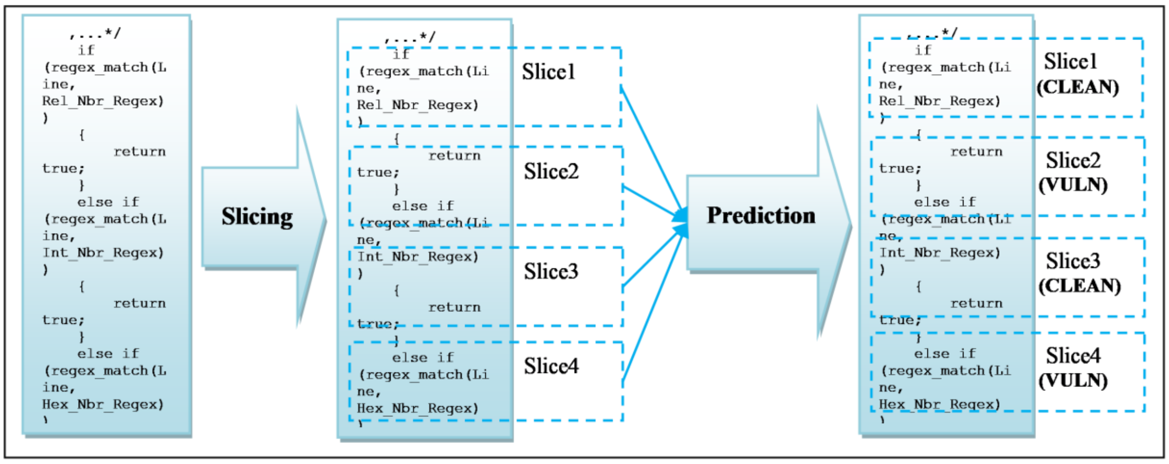
```
   ,...*/
    if
 (regex_match(Line,
 Rel_Nbr_Regex)
 )
      {
         return
 true;
      }
      else if
 (regex_match(Line,
 Int_Nbr_Regex)
 )
      {
         return
 true;
      }
      else if
 (regex_match(Line,
 Hex_Nbr_Regex)
 )
```

Figure 3. The adopted granularity level.

This way, only lines of code that are related to vulnerabilities can be extracted and analysed, which leads to indicate the exact location of vulnerabilities [22].

Currently, the slice level is the finest level of granularity to locate vulnerabilities. Predicting the status of a slice (clean or vulnerable) is just like locating the vulnerable lines (that the vulnerable slices contain) (Figure 3). Therefore, the slice level of granularity is adopted in this study.

3.2 Input Vectorization

As shown in Figure 3, a detection system built based on our proposed approach will take as input the source code of a software component (s) (file(s) or function(s)) and give in the output the vulnerable lines (those lines that compose the slices predicted as vulnerable). Since the prediction (deep features learning + classification) is made *via* DNNs, the source code of each extracted slice must be converted into vectors suitable to be used as input for DNNs (Figure 4, solid lines).

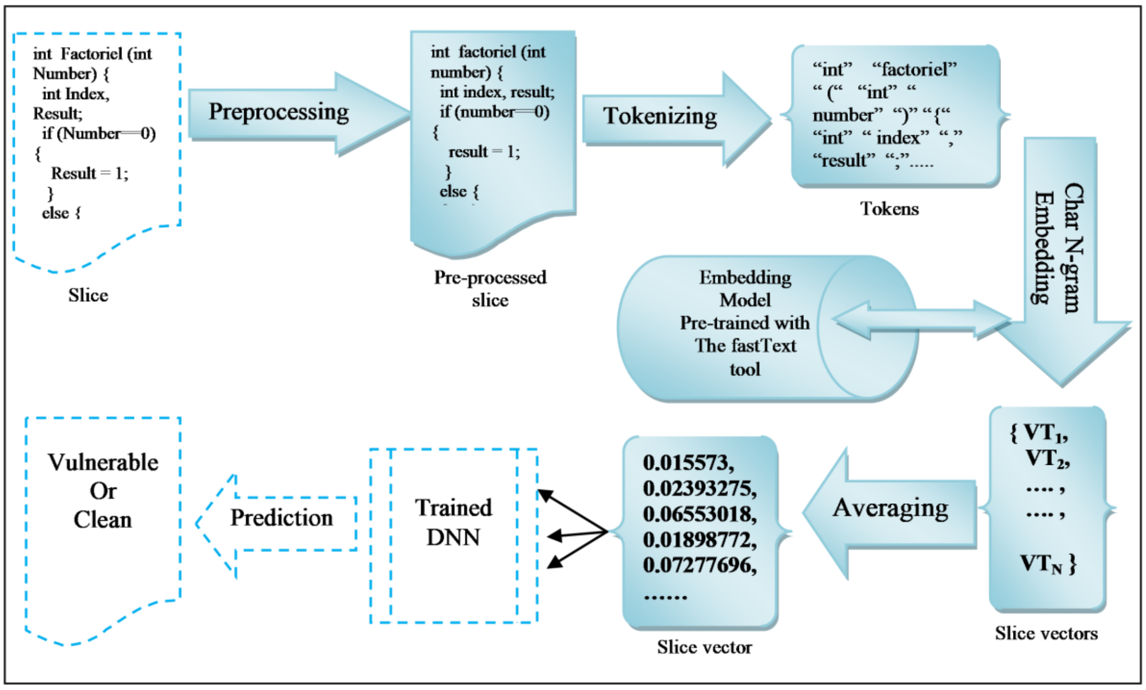
int Factoriel (int Number) { int Index, Result; if (Number==0) { Result = 1; } else {
Slice
Preprocessing
int factoriel (int number) { int index, result; if (number==0) { result = 1; } else {
Pre-processed slice
Tokenizing
"int" "factoriel" "(" "int" " number" ")" "{" "int" " index" "," "result" ";".....
Tokens
Char N-gram Embedding
Embedding Model Pre-trained with The fastText tool
{ VT1, VT2, …. , …. , VTN }
Slice vectors
Averaging
0.015573, 0.02393275, 0.06553018, 0.01898772, 0.07277696, ……
Slice vector
Trained DNN
Prediction
Vulnerable Or Clean

Figure 4. The proposed input vectorization approach (VTi means the vector representation of the ith token).

We aim to benefit from the power of embedding techniques to vectorize source code without losing useful semantic and syntactic information related to vulnerabilities induced by reducing vocabularies techniques. To achieve this aim, instead of reducing the vocabularies and applying token-level embedding as it was done in previous works (see related works section), we adopted solution [1], [36] proposed by the NLP community to deal with the embedding task in languages with large vocabularies that contain many rare words which is the case with source code. The strength of the solution proposed by [1] lies in using character n-gram-based embedding, which means that every token is embedded using all its character n-gram (sub-tokens). This has two advantages:

- Avoiding vocabulary reduction, because it is possible to embed almost any token using vectors of its sub-tokens.
- Enriching semantic information of tokens by the information of their sub-tokens.

In our approach, a very light pre-processing (removing comments and lowercasing the code) is made on the source code before transforming it into a set of tokens. Then, all the tokens are embedded using a pre-trained embedding model which gives a set of vectors. Because on one hand, the DNNs take as input vectors with fixed size and on the other hand slices may contain a variable number of tokens, we relied on the principle of sentence embedding to get a fixed-length that represents the slice (which is considered as a sentence). This can be done using a very simple approach, such as summing or averaging all token vectors of the sentence [39] or with sophisticated approaches, such as the approach proposed in [40]. We applied the simple approach by averaging all the token vectors of a slice to get its vector representation.

4. EXPERIMENTAL EVALUATION

In this section, we present the experiments conducted to evaluate our proposed approach. We begin by presenting the data preparation in 4.1 and then in 4.2, we present the implementation and architecture of the DNNs and finish by presenting the performance evaluation.

4.1 Data Preparation

To carry out the experiments, we prepared a dataset generated following the steps of the proposed input vectorization approach (Figure 4). The prepared dataset is then used to train and validate the DNNs to detect vulnerabilities. Before presenting our dataset, we begin by briefly describing the original dataset from which we retained the labeled source codes used to generate our dataset.

4.1.1 Slices Dataset

The labeled source codes from which we generated the dataset are retained from a dataset proposed by [28] which is publically available in [41]. The original dataset contains 420627 labeled slices, including 56395 vulnerable slices and 364232 clean slices. The slices were generated based on 1591 open-source C/C++ programs from the National Vulnerability Database (NVD) and 14,000 programs from the Software Assurance Reference Dataset (SARD). For the sake of brevity, the process of extracting and labeling the slices will not be described here. For more information about that process, see the slices dataset in [28]. We have chosen this dataset because it meets our needs:

- The source codes are organized inside the dataset in the form of labeled slices.
- Several types of vulnerabilities are considered (Library/API function call, array usage, pointer usage and arithmetic expression-related vulnerabilities).
- The dataset is very large, which makes it very suitable for DL techniques.
- On the contrary to other proposed datasets, the labeling process is based on real vulnerability reports mined from famous vulnerability databases, such as NVD, which is more efficient than using static analyzer tools to label source codes which can lead to mislabeling the source entities, because these tools can make high false-positive/negative reports.

4.1.2 Embedding Model Preparation

The proposed dataset contains two parts. The first part (vector dataset), which is used to train and validate the proposed DL-based VPMs, contains the vector representations and the class labels (Vulnerable/Clean) of each slice from the slices that exist in the original dataset. As shown in Figure 4, the vector representations are calculated based on the pre-trained embedding model. This model is prepared and trained using the second part of the dataset.

The data used to train the embedding model contains the tokenized slices (the tokens) without class labels. To prepare this data, we developed a C++ parser to parse the original dataset and do the following steps :

1. Extracting the slice.
2. Applying a minimalist pre-processing (removing comments if any).
3. Tokenizing the slice.
4. Composing a dataset line by gathering the tokens (separated by spaces) and adding it to the dataset.
5. Repeating the above steps for all the slices of the original dataset.

Using this data and the fastText tool [1], [36], we built the embedding model. The most important parameters of the model are its dimension (dim) and the range of size for the subwords (the minimum character n-grams size (minn) and the maximal character n-grams size (maxn)). The dimension controls the size of the vectors, where the larger they are, the more information they can capture, but the model requires more data to be learned. The range of size for the subwords controls the character n-grams that can be extracted from the tokens [42]. Building the embedding model with the defaults recommends that the values of these two parameters (dim:100, minn:3 and maxn:6) were very sufficient (see results section). However, since the size of the vectors also controls the architecture of

DNNs (the number of neurons in the input layer = the size of the vectors), we built two other models with the size of the vectors of 50 and 200. The other parameters related to training the models were also set to their default recommended values: 5 for the number of epochs and 0.05 for the learning rate.

4.1.3 Vectors Dataset (The Proposed Dataset)

To predict the status (vulnerable or clean) of a slice, its vector representation is extracted and passed to the trained DNN which learns more deep hidden features from the input vector and classifies them (Figure 4). To train the DNN, we prepared a labeled dataset that contains all the vector representations of the slices which exist in the original dataset with their class label (vulnerable or clean). As we mentioned before, the vectors are calculated based on the built embedding model. We developed a Java application to generate the dataset. The application is based on the Java implementation of the fastText API provided by the famous deep learning library Deeplearning4J [43]. The dataset contains a total number of 420627 instances, including 56395 instances with the 'vulnerable' class label and 364232 instances with the 'clean' class label. Detailed descriptive statistics by each type of vulnerabilities can be observed in Table 1.

Table 1. Descriptive statistics about the proposed dataset.

Type of vulnerabilities	Number of instances with class label (vulnerable)	Number of instances with class label (clean)	Total
Part1: Function Call (FC)	13603	50800	64403
Part2: Array Usage (AU)	10926	31303	42229
Part3: Pointer Usage (PU)	28391	263450	291841
Part4: Arithmetic Expression (AE)	3475	18679	22154
Total	56395	364232	420627

The embedding models and all the tools developed and used to generate them are made publically available in the public Github repository [44] for researchers who may want to replicate the study or to use it in other works.

4.2 DL-based VPM Construction and Evaluation

4.2.1 DL-based VPM Construction

We used the implementation of the Multi-Layer Perceptron (MLP) provided by the Java API of the WekaDeeplearning4J package [45] to construct our DL-based VPMs. This package provides a rich API inherited from the Weka API [46] and the Deeplearning4J library that facilitate not only the construction and the validation of a prediction model, but also the deployment of the built models in a concrete Java application to be used in production.

The effectiveness of the deep hidden features learning and consequently the overall prediction performance of the DL-based VPMs can be affected by several parameters related to the used DNN architecture: number of hidden layers, number of neurons in each layer, …etc. and the parameters related to the training process: learning rate, number of epochs, …etc. We considered all these parameters when conducting experiments. Some of these parameters, such as the learning rate, were set to their recommended values based on previous similar studies and what experts in the field of DL recommend. Other parameters, such as those related to the DNN architecture, were tuned experimentally. For example, when tuning the number and the size (in terms of the number of neurons) of the hidden layers, we started with simple architecture and each time we increased the complexity of the architecture, we observed the obtained results until we got the best performance.

4.2.2 VPM Evaluation

To accurately evaluate our DL-based VPMs and avoid the possibility of obtaining biased results, we used the technique of K-fold cross-validation to train and validate them. Using this technique, the dataset is randomly divided into K folds of equal sizes. K-1 folds are retained and used as the training set and the remaining fold is used as the testing set. This process is repeated such that all folds are used as the testing set and also as part of the training set. The final performance results are then calculated by averaging the results of all the iterations. Because our dataset is large enough, we set the value of K to 3. This allowed us to get very good results and reduce computation time.

A perfect VPM must have the following features:

- The VPM must predict as vulnerable only the actually vulnerable source entities. If this feature is not sufficiently achieved (i.e., the model leverages high false-positive predictions), the costs of vulnerability detection will not be minimized, since developers will still waste time and effort in looking for vulnerabilities in non-vulnerable source entities. This characteristic can be measured by the metric of the False Positives Rate (FPR).
- The VPM must not miss any vulnerability. This characteristic is very important, because if it is altered, vulnerabilities will be delivered with the software and can be exploited, which can lead to disastrous security issues. This characteristic can be measured by the metric of the False Negatives Rate (FNR).
- The VPM must make a precise and effective prediction. This characteristic can be measured by metrics, such as Precision or Recall.

Since obtaining a perfect VPM (FPR=0%, FNR=0% and Precision=100%) is impossible in practice, the objective is to minimize as much as possible the FPR and the FNR and to maximize as much as possible Precision. These metrics can be calculated from the outputs of the VPM: True Positive (TP), True Negative (TN), False Positive (FP) and False Negative (FN). The descriptions and the formulae of these metrics are shown in Table 2. For the sake of completeness, we also considered reporting in the results section, the obtained performance in terms of additional performance metrics that are communally reported in related works.

Table 2. The performance metrics.

Metric	Formula	Description
Precision	$\frac{TP}{TP+FP}*100$	The percentage of instances classified as positives and are actually positives.
FP Rate	$\frac{FP}{FP+TN}*100$	The percentage of positives that are falsely classified as real negatives.
FN Rate	$\frac{FN}{FN+TP}*100$	The percentage of negatives that are falsely classified as real positives.

5. RESULTS AND DISCUSSION

In Table 3, we report the obtained results in terms of the main performance metrics which we based our work on, in order to draw the conclusions: Precision (the higher the better), FPR and FNR (the lower the better). For the sake of completeness and for comparing the obtained results with the reported results in similar works, we report in Table 4 the obtained results in terms of additional performance metrics: Recall and F1 (the higher the better).

These results were obtained using a Multi-Layer Perceptron with the following architecture: 1 input layer with 100 neurons, 3 hidden layers with 128, 64 and 32 neurons and an output layer with 2 neurons. The MLP was trained and validated using the proposed dataset and 3-fold cross-validation. The parameters related to the training algorithm were set as follows (learning rate: 0,01, batch size:128) and the other parameters were set to their default recommended values (more information about these defaults recommended values can be found in the documentation of the Java API of Wekadeeplearning4j [47]). When carrying out the experiments, we begin by using the first part of the dataset which is related to the FC vulnerabilities to tune experimentally the DNN’s hyperparameter related to the number of epochs to train through and the size of the vectors for the dataset. We used only the FC part due to computational constraints. For the sake of showing the impact of these parameters on the VPM performance, we report results for different values of these parameters in the first rows of Table 3 and Table 4 for the number of epochs and in Table 6 for the size of the vectors.

Table 3. Results.

Vulnerabilities	No. of epochs	Precision (%)	FP Rate (%)	FN Rate (%)
FC	100	95,1	13,25	4,84
	500	96,8	8,30	3,12
	1000	**97,1**	**7,66**	**2,87**
AE	1000	98,0	7,48	1,93
AU	1000	97,1	6,36	2,86
PU	1000	**98,6**	**5,51**	**1,44**

Table 4. Results in terms of additional performance indicators.

Vulnerabilities	No. of epochs	F1 (%)	Recall (%)
FC	100	95,06	95,2
	500	96,84	96,9
	1000	**97,1**	**97,1**
AE	1000	**98.04**	**98,1**
AU	1000	97,1	97,1
PU	1000	98,55	98,6

As can be seen in Table 3 and Table 4, the obtained performances in terms of all the performance metrics (whether the main ones or the additional ones) and for all the studied types of vulnerabilities (FC, AE, AU and PU) are very excellent. Indeed, the obtained precision was between 97,1% and 98,6%, the FNR was between 2,87% and 1,44% and the FPR was between 7,66% and 5,51%, which is very promising and outperformed the obtained performances in the previous works that used the same original dataset and the same granularity level (slice) ([16], [29]) and others ([18], [27]) that used different datasets and different granularity levels (Table 5). We believe that this performance improvement is due to the two strengths of the proposed input vectorization approach, which were missing in the previous approaches. The first is embedding the source code without reducing the vocabulary, which led to preserving all semantic and syntactic information of the source code related to vulnerabilities. The second is that this information is enriched by embedding source code in character n-gram level, because each token is embedded using its sub-tokens. For example, considering a token named "BufferSize", using the technique of character n-gram embedding, this token will be embedded using all its character n-gram, including "Buffer" and "Size", which means that two important things are granted. The first is that even if the token "BufferSize" does not exist in the training set, it will be possible to get its vector representation from its sub-tokens vector representations. The second is that the overall semantic meaning of the original token "BufferSize" will be enriched by the semantic meaning of its sub-tokens, including "Buffer" and "Size".

We observed that the obtained values in terms of FPR were slightly higher when compared to the obtained values in terms of FNR (5,51% vs 1,44%). We confirm what other researchers [29] concluded about this situation. Indeed, improving the performance of a VPM in terms of one of these two metrics can affect its performance in terms of the other. As we mentioned before and as it is clear in Table 6, our obtained results outperformed the reported results of all the previous studies in terms of all performance metrics, except in terms of FPR. While we obtained in the best case 5,51%, researchers in [29] reported a value of 1,4%. The reported value is better than what we got, but it comes at the cost of much higher FNR (5,6%) than what we got (1,44%). In vulnerability prediction, the negative impact of FNR is much important than the negative impact of FPR. This is because FPR impacts the effectiveness of the model in terms of the cost of detection, while FNR impacts its effectiveness in the side of letting vulnerabilities undiscovered, which is very dangerous. Therefore, we believe that the advantages in terms of a lower FNR far outweigh the disadvantages concerning the slightly higher FPR.

We observed also that all the best values for all the experimental cases were obtained using the part of the dataset that is related to PU vulnerabilities. Since this part is the larger one in the dataset (see Table1), we believe that this was due to the sufficient amount of data that this part contains, which let the DNN learn more efficiently than with the other parts. This lets us conclude that sufficient labeled data is very important when using the DL technique to predict vulnerabilities.

Table 5. Comparison with the reported results of previous studies.

Study	**Granularity level**	**Vocabulary reducing**	**Vectorization / Technique**	**Performances (%)**
[16]	Slice	yes	word2vec-based techniques / (Token level)	-FPR: 5,7 -FNR: 7,0 -P: 88,1 -R: / -F1: 90,5
[29]	Slice	yes	word2vec-based techniques / (Token level)	**-FPR: 1,4** -FNR: 5,6 -P: 90,8 -R: / -F1: 92,6
[27]	Function	yes	Custom embedding technique / (Token level)	-FPR: / -FNR: / -P: / -R: / -F1: 84,0
[18]	File	yes	Custom embedding technique / (Token level)	-FPR: / -FNR:/ -P: 92,0 -R:93,0 -F1 :91,0
Our Study	**Slice**	**no**	**fastText-based embedding / (Character n-gram level)**	-FPR : 5,51 **-FNR :1,44** **-P :98,6** **-R :98,6** **-F1 : 98,55**

Table 6. Results for different vector sizes.

Vector size	**Precision (%)**	**FP Rate (%)**	**FN Rate (%)**	**Recall (%)**	**F1 (%)**
50	96,3	09,37	03,68	96,3	96,27
100	97,1	7,66	2,87	97,1	97,1
200	**97,3**	**07,49**	**02,67**	**97,3**	**97,29**

To investigate the impact of the size of the vectors on the VPM performances, we pre-trained 3 embedding models with 3 different values of the vector size parameter: 50,100 and 200. Then, we used the pre-trained models to prepare 3 datasets. The prepared datasets were then used to train and validate 3 VPMs. The obtained results are shown in Table 6. As can be seen, there were no significant differences between the 3 VPMs performances. This can be interpreted by the fact that models with higher vectors size (>200) need very big training data to capture much useful vector representation. Finally, we can conclude that the difference in performances (Precision: +0.2%, FPR: -0,17%, FNR: -0,2%) obtained by increasing the vector size to 200 is not worth the constrains in terms of increasing the model size induced by increasing the size of the vectors. That way in this study, we opted for 100 as the size of the vectors.

5.1 Limitations

We are aware that our work may have the following limitations:

- Since the original dataset from which we generated the data used to evaluate the proposed approach is limited to C/C++ code, we cannot conclude that the approach is suitable for other types of applications that are written in other languages. Evaluating the proposed approach for these types of applications represents an intersecting open problem for future research works.
- As the focus of the study was on proposing an input vectorization approach, we used only one DNN model, the MLP. Even though the obtained results using this model were very sufficient, other DNN models, such as RNN and CNN, must be considered in future works.
- We evaluated our approach for 4 types of vulnerabilities using binary classification. A multiclass classification will be a better choice for future works.
- More types of vulnerabilities must be considered in future works.

6. Conclusions

This paper investigated predicting vulnerabilities using the technique of deep learning. We aimed at improving the communally adopted approach in recent similar works which was inspired by the previous application of DL in NLP. Our contribution was to propose an efficient input vectorization approach based on embedding source code in the character n-gram level. Using the proposed approach with DNNs, we were able to efficiently learn deep hidden features and detect vulnerabilities with much better accuracy. The strengths of our method lie in preserving and enriching semantic and syntactic information related to vulnerabilities that can be extracted from the code. Indeed, the achieved performance outperformed those obtained in previous similar works, which means that our method represents a valuable alternative to the input vectorization methods used in previous works.

As part of this research, we proposed a dataset extracted from a labeled and large C/C++ codebase. The dataset was prepared based on the proposed input vectorization approach. We make it with other important data publically available for the community. As part of the future works and since the results obtained have been very promising, we plan to implement the proposed approach in a concrete solution that can be used in production. The solution can be in the form of a standalone AVP tool or an IDE plug-in. Improving further the proposed approach by addressing the limitations and the open research problems indicated in the previous section and considering more code structure for learning more comprehensive program semantics that is suitable for AVP, represent subjects for interesting future works a well.

Acknowledgements

The authors would like to thank the anonymous reviewers for their insightful comments and suggestions that helped in improving the paper.

References

[1] P. Bojanowski, E. Grave, A. Joulin and T. Mikolov, "Enriching Word Vectors with Subword Information," Trans. Assoc. Comput. Linguist., vol. 5, pp. 135–146, DOI: 10.1162/tacl_a_00051, 2017.

[2] Y. Shin, A. Meneely, L. Williams and J. A. Osborne, "Evaluating Complexity, Code Churn and Developer Activity Metrics As Indicators of Software Vulnerabilities," IEEE Trans. Softw. Eng., vol. 37, no. 6, pp. 772–787, DOI: 10.1109/TSE.2010.81, 2011.

[3] T. Zimmermann, N. Nagappan and L. Williams, "Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista," Proc. of the 3rd Int. Conf. on Software Testing, Verification and Validation (ICST 2010), pp. 421–428, DOI: 10.1109/ICST.2010.32, Paris, France, 2010.

[4] P. Morrison, K. Herzig, B. Murphy and L. Williams, "Challenges with Applying Vulnerability Prediction Models," Proceedings of the 2015 Symposium and Bootcamp on the Science of Security (HotSoS '15), pp. 1–9, DOI: 10.1145/2746194.2746198, 2015.

[5] S. Moshtari and A. Sami, "Evaluating and Comparing Complexity, Coupling and a New Proposed Set of Coupling Metrics in Cross-project Vulnerability Prediction," Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC '16), pp. 1415–1421, DOI: 10.1145/2851613.2851777, 2016.

[6] I. Abunadi and M. Alenezi, "Towards Cross Project Vulnerability Prediction in Open Source Web Applications," Proceedings of the the International Conference on Engineering & MIS 2015 (ICEMIS '15), pp. 1–5, DOI: 10.1145/2832987.2833051, 2015.

[7] J. Walden, J. Stuckman and R. Scandariato, "Predicting Vulnerable Components: Software Metrics *vs.* Text Mining," Proc. of the 25th IEEE International Symposium on Software Reliability Engineering (ISSRE), pp. 23–33, DOI: 10.1109/ISSRE.2014.32, Naples, Italy, 2014.

[8] M. Zagane and M. K. Abdi, "Evaluating and Comparing Size, Complexity and Coupling Metrics As Web Applications' Vulnerabilities Predictors," Int. J. Inf. Technol. Comput. Sci., vol. 11, no. 7, pp. 35–42, DOI: 10.5815/ijitcs.2019.07.05, 2019.

[9] M. Zagane, M. K. Abdi and M. Alenezi, "A New Approach to Locate Software Vulnerabilities Using Code Metrics," Int. J. Softw. Innov., vol. 8, no. 3, pp. 82–95, DOI: 10.4018/IJSI.2020070106, Jul. 2020.

[10] A. Hovsepyan, R. Scandariato, W. Joosen and J. Walden, "Software Vulnerability Prediction Using Text Analysis Techniques," Proceedings of the 4th International Workshop on Security Measurements and Metrics (MetriSec '12), p. 7, DOI: 10.1145/2372225.2372230, 2012.

[11] B. Turhan and A. Bener, "A Multivariate Analysis of Static Code Attributes for Defect Prediction," Proceedings of the 7th IEEE International Conference on Quality Software (QSIC 2007), pp. 231–237, DOI: 10.1109/QSIC.2007.4385500, 2007.

[12] H. Abandah and I. Alsmadi, "Call Graph Based Metrics to Evaluate Software Design Quality," Int. J. Softw. Eng. and Its Appl., vol. 7, no. 1, pp. 1–12, 2013.

[13] T. Hall, S. Beecham, D. Bowes, D. Gray and S. Counsell, "A Systematic Literature Review on Fault Prediction Performance in Software Engineering," IEEE Transactions on Software Engineering, vol. 38, no. 6. pp. 1276–1304, DOI: 10.1109/TSE.2011.103, 2012.

[14] B. Turhan, A. Bener and T. Menzies, "Nearest Neighbor Sampling for Cross Company Defect Predictors," Proceedings of the 1st International Workshop on Defects in Large Software Systems (DEFECTS'08), p. 26, DOI: 10.1145/1390817.1390824, 2008.

[15] T. Menzies, J. Greenwald and A. Frank, "Data Mining Static Code Attributes to Learn Defect Predictors," IEEE Trans. Softw. Eng., vol. 33, no. 1, pp. 2–14, DOI: 10.1109/TSE.2007.10, 2007.

[16] Z. Li et al., "VulDeePecker: A Deep Learning-based System for Vulnerability Detection," Proceedings of Network and Distributed System Security Symposium, DOI: 10.14722/ndss.2018.23158, 2018.

[17] T. Shippey, D. Bowes and T. Hall, "Automatically Identifying Code Features for Software Defect Prediction: Using AST N-grams," Inf. Softw. Technol., vol. 106, pp. 142–160, DOI: 10.1016/j.infsof.2018.10.001, Feb. 2019.

[18] H. K. Dam, T. Tran, T. T. M. Pham, S. W. Ng, J. Grundy and A. Ghose, "Automatic Feature Learning for Predicting Vulnerable Software Components," IEEE Trans. Softw. Eng., pp. 1–1, DOI: 10.1109/TSE.2018.2881961, 2019.

[19] C. Catal, "Can We Predict Software Vulnerability with Deep Neural Network ?" Proc. of the 19th Int. Multiconference INFORMATION SOCIETY- IS, no. October, pp. 19–22, Ljubljana, Slovenia, 2016.

[20] C. Catal, A. Akbulut, E. Ekenoglu and M. Alemdaroglu, "Development of a Software Vulnerability Prediction Web Service Based on Artificial Neural Networks," Proc. of the Pacific-Asia Conference on Knowledge Discovery and Data Mining, pp. 59–67, DOI: 10.1007/978-3-319-67274-8_6, 2017.

[21] J. Walden, J. Stuckman and R. Scandariato, "Web Apps Vulnerability Dataset," [Online], Available: http://seam.cs.umd.edu/webvuldata, 2014.

[22] M. Zagane, M. K. Abdi and M. Alenezi, "Deep Learning for Software Vulnerabilities Detection Using Code Metrics," IEEE Access, vol. 8, pp. 74562–74570, DOI: 10.1109/ACCESS.2020.2988557, 2020.

[23] M. Zagane and M. K. Abdi, "Code Mmetrics Dataset (PU)," [Online]. Available: https://github.com/codemetricsdaset/slice_codemetricsdataset/.

[24] F. Tip, "A Survey of Program Slicing Techniques," J. Program. Lang., vol. 5399, no. 3, pp. 1–65, 1995.

[25] M. Weiser, "Program Slicing," IEEE Trans. Softw. Eng., vol. SE-10, no. 4, pp. 352–357, DOI: 10.1109/TSE.1984.5010248, Jul. 1984.

[26] J. Silva, "A Vocabulary of Program Slicing-based Techniques," ACM Comput. Surv., vol. 44, no. 3, pp. 1–41, DOI: 10.1145/2187671.2187674, Jun. 2012.

[27] R. Russell et al., "Automated Vulnerability Detection in Source Code Using Deep Representation Learning," Proceedings of the 17th IEEE International Conference on Machine Learning and Applications (ICMLA), pp. 757–762, DOI: 10.1109/ICMLA.2018.00120, Orlando, USA, 2019.

[28] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu and Z. Chen, "SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities," arXiv:1807.06756v2, pp. 1–13, DOI: 10.21227/fhg0-1b35, Jul. 2018.

[29] Z. Li, D. Zou, J. Tang, Z. Zhang, M. Sun and H. Jin, "A Comparative Study of Deep Learning-based Vulnerability Detection System," IEEE Access, vol. 7, pp. 103184–103197, DOI: 10.1109/ACCESS.2019.2930578, 2019.

[30] S. Liu et al., "CD-VulD: Cross-Domain Vulnerability Discovery Based on Deep Domain Adaptation," IEEE Trans. Dependable Secur. Comput., pp. 1–1, DOI: 10.1109/TDSC.2020.2984505, 2020.

[31] T. Mikolov, K. Chen, G. Corrado and J. Dean, "Efficient Estimation of Word Representations in Vector Space," Proc. of the 1st International Conference on Learning Representations (ICLR 2013), arXiv:1301.3781v3, [Online], Available: https://storage.googleapis.com/pub-tools-public-publication-data/pdf/41224.pdf, 2013.

[32] C. Tomas Mikolov, "Word2Vec.," Google Inc., Mountain View, [Online], Available: https://code.google.com/archive/p/word2vec/.

[33] Y. Kim, "Convolutional Neural Networks for Sentence Classification," Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 1746–1751, DOI: 10.3115/v1/D14-1181, Doha, Qatar, 2014.

[34] S. Liu, G. Lin, Q.-L. Han, S. Wen, J. Zhang and Y. Xiang, "DeepBalance: Deep-Learning and Fuzzy Oversampling for Vulnerability Detection," IEEE Trans. Fuzzy Syst., pp. 1–1, DOI: 10.1109/TFUZZ.2019.2958558, 2019.

[35] X. Ban, S. Liu, C. Chen and C. Chua, "A Performance Evaluation of Deep-learnt Features for Software Vulnerability Detection," Concurrency Computation, vol. 31, no. 19, DOI: 10.1002/cpe.5103, 2019.

[36] T. M. P. Bojanowski, E. Grave and A. Joulin, "fastText," Library for Efficient Text Classification and Representation Learning, [Online], Available: https://fasttext.cc/.

[37] X. Du et al., "LEOPARD: Identifying Vulnerable Code for Vulnerability Assessment through Program Metrics," Proceedings of the 41st International Conference on Software Engineering (ICSE '19), vol. 2019-May, pp. 60–71, DOI: 10.1109/ICSE.2019.00024, Jan. 2019.

[38] K. Pan, S. Kim and E. Whitehead, Jr., "Bug Classification Using Program Slicing Metrics," Proc. of the 6th IEEE International Workshop on Source Code Analysis and Manipulation, pp. 31–42, DOI: 10.1109/SCAM.2006.6, 2006.

[39] J. Wieting, M. Bansal, K. Gimpel and K. Livescu, "Towards Universal Paraphrastic Sentence Embeddings," Proc. of the 4th International Conference on Learning Representations (ICLR 2016), pp. 1-19, [Online], Available: https://arxiv.org/pdf/1511.08198.pdf, 2016.

[40] S. Arora, Y. Liang and T. Ma, "A Simple But Tough-to-beat Baseline for Sentence Embeddings," Proc. of the 5th International Conference on Learning Representations (ICLR 2017), pp. 1-16, [Online], Available: https://openreview.net/pdf?id=SyK00v5xx, 2019.

[41] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu and Z. Chen, "SeVC and SyVC Dataset," [Online], Available: https://github.com/SySeVR/SySeVR/.

[42] T. M. P. Bojanowski, E. Grave, A. Joulin, "fastText Documentation," [Online], Available: https://fasttext.cc/docs/.

[43] DL4J, "Deep Learning for Java," [Online], Available: https://deeplearning4j.org/, 2020.

[44] GitHub, "Char N-gram Embedding Dataset for DL-based AVP," [Online], Available: https://github.com/dzresearcher/char_n-gram_embedding_dataset_for_DL_AVP.

[45] S. Lang, F. Bravo-Marquez, C. Beckham, M. Hall and E. Frank, "WekaDeeplearning4j: A Deep Learning Package for Weka Based on Deeplearning4j," Knowledge-Based Syst., vol. 178, pp. 48–50, DOI: 10.1016/j.knosys.2019.04.013, Aug. 2019.

[46] Machine Learning Group at the University of Waikato, "Weka API Online Doc," [Online], Available: http://weka.sourceforge.net/doc.dev/.

[47] GitHub, "Online Documentation of the Wekadeeplearning4j Java API," [Online], Available: https://waikato.github.io/wekaDeeplearning4j/.

ملخص البحث:

لقد تم بنجاحٍ تطبيق تقنيات التّعلُّم العميق لحل المشكلات التي تشكل تحدياتٍ في مجال معالجة اللغات الطبيعيّة. ونظراً لأن رمز المصدر والنص الطبيعي يشتركان في عدد من الأمور المتشابهة، فقد أمكن اعتماد تقنيات تصنيف النصوص - مثل تضمين الكلمات - لاقتراح طرقٍ تستند على التعلُّم العميق للقيام بالتّنبؤ الآلي بالثغرات البرمجية. وعلى الرغم من أن النتائج كانت مثيرة للاهتمام، فلم تكن بالجودة الكافية مقارنةً بتلك التي تم الحصول عليها في معالجة اللغات الطبيعية. وفي هذه الورقة، نقترح طريقة محسنة قائمة على التنبؤ الآلي بالثغرات البرمجية بناءً على تقنية تضمين الرموز (ن-غرام). وقد عملنا على تقييم الطريقة المقترحة لأربعة أنواع من الثغرات البرمجية مستخدمين قاعدة رموز ضخمة ذات مصدر مفتوح بلغة C/C++. وقد أظهرت النتائج أن الطريقة المقترحة تتمتع بأداءٍ جيد للغاية يتفوق على كثيرٍ من الطرق الواردة في دراساتٍ سابقة.