IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 11, 2020, accepted April 6, 2020, date of publication April 17, 2020, date of current version May 4, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2988557

Deep Learning for Software Vulnerabilities Detection Using Code Metrics

MOHAMMED ZAGANE[1], MUSTAPHA KAMEL ABDI[1], AND MAMDOUH ALENEZI[2], (Member, IEEE)

[1]Département d'informatique, Université Oran1, Oran 31000, Algeria
[2]Department of Computer Science, College of Computer and Information Sciences, Prince Sultan University, Riyadh 12435, Saudi Arabia

Corresponding author: Mohammed Zagane (m_zagane@esi.dz)

This work was supported by the College of Computer and Information Sciences, Prince Sultan University, Riyadh, Saudi Arabia.

ABSTRACT Software vulnerability can cause disastrous consequences for information security. Earlier detection of vulnerabilities minimizes these consequences. Manual detection of vulnerable code is very difficult and very costly in terms of time and budget. Therefore, developers must use automatic vulnerabilities prediction (AVP) tools to minimize costs. Recent works on AVP begin to use techniques of deep learning (DL). All the proposed approaches are based on techniques of feature extraction inspired by previous applications of DL such as automatic language processing. Code metrics were widely used as features to build AVP models based on classic machine learning. This study bridges the gap between deep learning and machine learning features and discusses a deep-learning-based approach to finding vulnerabilities in code using code metrics. Obtained results show that code metrics are very good but not the better to use as features in DL-based AVP.

INDEX TERMS Automatic vulnerability prediction, code metrics, deep neural networks.

I. INTRODUCTION

The presence of vulnerabilities in software is inevitable because writing secure code is very difficult and requires a lot of expertise. And since humans are prone to mistake, even experienced and elite developers can make programming errors that lead to disastrous consequences on information security. Earlier detection of software vulnerabilities minimizes these consequences. The manual detection of vulnerable code is very difficult, tedious task and very costly in terms of time and budget. To assist developers and minimize these costs, tools that can automatically predict vulnerable source entities (file, function, etc.) must be used to let developers focus their efforts on most likely vulnerable components. These costs can be minimized even more if the used tools can identify the exact location of vulnerabilities (vulnerable source lines). In the research field, the automatic vulnerability prediction (AVP), drew the attention of many researchers. Indeed, many approaches of AVP have been proposed in the literature such as vulnerability prediction models (VPMs) built based on machine learning (ML) techniques and software features (software metrics, bag-of-words, etc.).

The associate editor coordinating the review of this manuscript and approving it for publication was Junaid Arshad.

Most of the proposed VPMs used software metrics as input to discriminate between vulnerable and clean source entities. Because in one hand, software metrics can perfectly quantify software characteristics such as complexity, coupling, and size and on the other hand, it was proven in the practice that there is a correlation between these software characteristics and vulnerabilities. The main limitation of the VPMs built using software metrics is that they do not have the capacity of pinpointing the exact location of vulnerabilities because metrics are calculated at a coarse level of granularity (package, file, class, function). Instead, they only predict if the source entity is vulnerable or clean.

In recent studies (see Related Works sub-section), researchers begin to investigate AVP using techniques of deep learning. The researchers used techniques inspired by previous applications of DL such as automatic language processing to automatically extract features. Although software metrics were widely used in previous studies that used classic ML in AVP, software metrics have not yet been evaluated as features in DL-based AVP. At the best of our knowledge, until now no one tried to use software metrics with DL to detect vulnerabilities (prediction at a finer granularity).

In this study, we aim to fill this research gap and to propose an approach of vulnerability detection based on DL and software metrics. As we mentioned in previous paragraphs,

software metrics can perfectly quantify software characteristics that are correlated to vulnerabilities. Therefore, they may represent good features to build VPMs based on DL.

The contribution of this study is twofold:

- Investigating the usefulness of code metrics as features for deep learning in vulnerability detection.
- Proposing a dataset of code metrics generated from labeled code slices: as part of the study, we propose and make publicly available a dataset of code metrics generated from labeled code slices.

The remainder of the paper is organized as follows: Section II presents background and the most relevant related works, Section III presents the research question and describes the proposed approach and the followed methodology to carry out the study, Section IV presents the experimentations, Section V presents the obtained results and discussion, Section VI presents the limitations of the study, Section VII summarizes the work done in this study and indicates some perspectives.

II. BACKGROUND AND RELATED WORK

In this section, background and most related works are briefly presented. We begin by giving the context of the study by presenting all concepts related to it (deep neural networks, vulnerability prediction, and code slicing) after that we terminate by citing most related works.

A. DEEP NEURAL NETWORKS

Deep learning is a subfield of ML. Most deep learning techniques are based on artificial neural networks (ANN). An ANN is a network of computational models: artificial neurons (ANs) that are inspired by the biological neuron of the human brain. The AN receives inputs from many other ANs, process them and produce an output that is then transmitted to other neurons. The data processing that occurs in each AN is very simple, but the global behavior of the whole network generated by the interaction of its ANs leads to solving complex problems [1].

Although the first attempts to model the brain are old (the 1940s), the interesting results were obtained just in the few recent years when deep ANNs (DNNs) were proposed. The ANs in a DNN are ranged in many interconnected layers. There are three types of layers:

- Input layer: There is only one input layer in a DNN. It receives inputs (the actual inputs of the global system which are also the data to be processed), processes them and transmits them to the hidden layers.
- Hidden layers: There are zero or several hidden layers in a DNN. The hidden layers let the DNN learn more about the data to be processed. The more the DNN has hidden layers, the more it can solve a complex problem. The outputs of each hidden layer become the input of the next hidden layer and so on.
- Output layer: Like the input layer, there is only one output layer in a DNN. It receives inputs from the last hidden layer, processes them and transmits them to the final output of the DNN which are also the final results generated by the system.

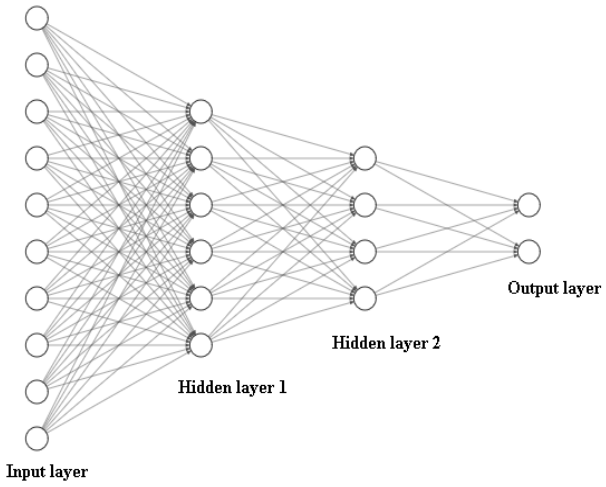

FIGURE 1. Simple DNN.

Figure 1 shows an example of a simple DNN that has the following architecture: input layer (10 neurons), two hidden layers (6 neurons and 4 neurons) and an output layer (2 neurons).

Just like classic ML algorithms, a DNN must be learned before using it. Supervised learning is the most used in applications such as pattern recognition, automatic language processing, and general classification problems. The supervised learning in DL needs large amounts of labeled data and powerful machines to run the learning process. For example, to build a face detector, one needs a large collection of images labeled as containing faces, often with a bounding box around the face [2]. That is why just in recent years that the DL has seen great success because of the growing computing power offered by the recent computers and the large amounts of labeled data offered by the new technologies such as the internet.

B. VULNERABILITY PREDICTION MODEL (VPM)

Given that software vulnerability is a particular kind of software defect that affects information security, techniques used in automatic defect prediction are also used in automatic vulnerability prediction (AVP). One of these techniques is Defect Prediction Models (DPM) [3]–[7]. DPMs are models predicting the existence or likelihood of code defects in code entities: file, class, function, etc. [8].

VPMs and DPMs are built based on machine learning techniques and using software attributes as input to discriminate between vulnerable and clean source entities [9]. The Data used to train and validate VPMs are collected from previous versions of the studied software. Based on vulnerabilities information extracted from vulnerability databases such as NVD (National Vulnerability Database), source entities are retained and labeled as vulnerable or clean. Then, features (software metrics) of each source entity are extracted and

collected in datasets. In cross-project prediction, datasets collected from other software are used. This type of prediction is useful when studying new software that does not have earlier versions or sufficient data on their vulnerabilities.

Most of the prior studies in the field of AVP tried to evaluate theories that are a correlation between software characteristics: complexity, coupling, etc and vulnerabilities [10]–[13]. In other studies [8], [14] researchers reported that the classic software metrics used in DPM are not accurate for VPM. They suggested that new security-specific code metrics must be used to improve VPM's performances. Others [15], [16] concluded that the main problem of VPMs is the limited data on vulnerabilities.

The proposed VPMs allow vulnerability prediction at a coarse granularity: package, file, class, and function. This can assist developers and let them focus their effort on most likely vulnerable source entities. However, if the source entities have a lot of lines of code, the task of pinning down the location of vulnerable pieces of code will be very difficult and time-consuming. Therefore, researchers proposed in recent works [17], [18] of vulnerability detection to use finer granularity, code gadgets where only a few lines of code that are related to vulnerabilities are involved. This notion of code gadget was inspired by the concept of code slicing. In the next sub-section, we will present this concept.

C. CODE SLICING

Code slicing or program slicing was first introduced by M. Weiser [19]. He defined program slicing as a method for automatically decomposing programs by analyzing their data flow and control flow to produce a reduced program called a slice. Since then, many research works were addressed the program slicing to propose: algorithms of computing slices, different types of code slicing and many applications in software engineering. A good description of features, main applications and a common example of slicing techniques can be found in [20].

The strong point of code slicing is that it can give insight about multiple behavioral aspects of the source entity, such as all source lines that change the value of a variable, or statements that participate in computing the return value of a function [21]. In the security context, this can be useful to get for examples: all statements that are related to a critical system call or all statement that can change the values of variables which are used as parameters in critical functions calls (memory management, string manipulation, etc.). This way, only lines of code that are related to vulnerabilities can be extracted and analyzed which lead when applied in vulnerability prediction to indicate the exact location of vulnerabilities.

D. RELATED WORK

For the sake of brevity, in this subsection, we present only the most relevant related works that investigated vulnerabilities detection using deep learning techniques.

Most of the prior research work in the field of AVP has only focused on classic machine learning techniques. In recent years, a few researchers begin to investigate AVP using techniques of deep learning. The first attempt to investigate deep learning in vulnerability prediction was done by [22]. Researchers did a literature review and concluded that depend on the previous vulnerability data, different kind of deep learning algorithms can be applied in AVP: supervised learning models if there is sufficient data, unsupervised deep learning models if there is no vulnerability data and semi-supervised models if there is a limited data. Since then, a few but very interesting studies have been carried out. These recent studies tried to make deep learning suitable for AVP by drawing inspirations by previous applications of deep learning techniques; where challenging problems such as pattern recognition, natural language processing, image processing, etc. were solved.

Inspired by the notion of region proposal in image processing, researchers in [17] proposed to divide a program into smaller pieces of code (i.e., number of statements), which may exhibit the syntax and semantics characteristics of vulnerabilities. These small pieces of code called code gadget (CG) can be obtained by extracting code slices (see the previous sub-section) from the source code. This also lets them predict vulnerability at a fine granularity (CG) which means that the exact location of vulnerabilities can be indicated by the proposed vulnerabilities detection system called VulDeePecker. To address the limitations of VulDeePecker (considering only the vulnerabilities that are related to library/API function calls, leveraging only the semantic information induced by data dependency and considering only the particular RNN known as Bidirectional Long Short-Term Memory (BLSTM)), researchers proposed in [18] a framework for using deep learning to detect vulnerabilities dubbed Syntax-based, Semantics-based, and Vector Representations (SySeVR). While in VulDeePecker only one RNN model was used, in SySeVR several models were evaluated (CNN, DBN, and 4 variants of RNN: LSTM, GRU, BLSTM, and BGRU).

Since DNNs take vectors as input, researchers proposed to transform program slices into vectors of tokens that are semantically meaningful for vulnerability detection. Based on the technique of word embedding, they encoded the tokens of each slices using the word2vect tool [23] into size-fixed vectors which are the actual input of the DNNs. This technique (word embedding) is inspired by the application of deep learning in natural language processing. In [24] researchers investigated two deep neural network models: CNNs and RNNs in vulnerabilities prediction in two ways:

- They used them to learn features based on the technique of word (or token) embedding. After that, the learned features are used as input to build a VPM based on a Random Forest classifier,
- They used them to learn features and as classifiers.

The studies that used automatically-learned features as inputs for DNNs have reported interesting results Table 1.

TABLE 1. Recent studies conducted in vulnerabilities detection using deep learning.

Study	Granularity level	DNN Models	Training Data	Reported results
[17]	slice	RNNs	automatically-learned features	-FPR: 5.7 -FNR: 7.0 -P: 88.1 -F1: 90.5
[18]	slice	CNNs, DBNs, RNNs	automatically-learned features	-FPR: 1.4 -FNR: 5.6 -P: 90.8 -F1: 92.6
[24]	function	CNNs, RNNs	automatically-learned features	-PR AUC: 94.4 -ROC AUC: 95.4 -MCC: 69.8 -F1: 84.0
[25]	file	MLP	Code metrics	-AUC: 76.5

However, this approach does not consider important code characteristics such as complexity which correlate with vulnerabilities [11], [14], [26]. Another limitation of this approach lies in the fact that on the one hand neural networks take a size-fixed vector as input, but on the other hand, the token number in each source entity (slice or function) may be different. To address this problem, in this approach a threshold is used. When a vector is shorter than the threshold, zeroes are padded to the end of the vector. When a vector is longer than the threshold, portions from the vector are deleted to make it equal to the threshold which may lead to a very important loss of information.

Code character ristics such as complexity can be quantified by corresponding code metrics. While several previous studies [8], [12]–[15], [26]–[28] have used software metrics as features to build VPM based on classic machine learning techniques, only one study [25] has used software metrics with deep learning to predict vulnerabilities at the file granularity level. In that study, authors proposed a web-service-based VPM to predict vulnerable components (files) of web applications, several machine learning techniques which exist in Azure Machine Learning Studio environment (Bayes point machine, Boosted decision tree, Random forest,. . .) and a multi-layer perceptron (MLP) were investigated. The authors reported that the best performance is achieved by the MLP (Last line in table 1). The main limitation of this study is that the prediction was at a coarse granularity (file granularity). This means that code metrics were calculated from the whole source code of the file (vulnerable lines and clean lines) and the built VPMs only predict if the file is vulnerable or clean without locating the vulnerable lines. Another limitation is that the used dataset [15] was proposed to evaluate VPM based on classic machine learning and does not provide sufficient data that let deep learning models achieve the best training. Therefore, in this work, we propose to predict vulnerabilities at the slice granularity. Instead of calculating metrics from the whole source code of the file or function, we extract slices (a few lines of code related to vulnerabilities) then we calculate metrics for each slice. This way, the built VPMs can predict if a slice (few lines of code) is vulnerable or clean which leads to locate vulnerable lines. We also use a large dataset (> 95351 instances) suitable for DL.

III. METHODOLOGY

In this section, we present the research questions, proposed approach and the different steps followed to conduct the study.

A. RESEARCH QUESTIONS

Motivated by the success of DL techniques in other fields such as pattern recognition, natural language processing, etc., the recent research works in the field of vulnerabilities prediction tends to use these techniques to predict vulnerabilities. Although code metrics were widely used and evaluated as features to build VPM based on classic machine learning techniques, the recent studies that used the technique of DL have only focused on automatically-learned features inspired by the previous applications of DL to build VPM. No one of them to the best of our knowledge has investigated whether code metrics can be used as features to build VPM based on these advanced techniques of DL.

The main aim of this study is to fill this research gap. To achieve this objective, this study will try to answer the following main research question:

RQ: Since code metrics were successfully used as features to build VPM based on classic machine learning techniques, can they be used as features with deep learning to detect vulnerabilities?

B. PROPOSED APPROACH

We proposed an approach inspired of a recent study that used deep neural networks to predict vulnerability at the slice granularity [17]. In the proposed approach (Figure 1), Instead of automatically learn features as it was done in the previous approaches [17], [18], [24] which is a technique inspired by the previous applications of DNN (natural language processing, patterns recognition, etc.), we use a set of code metrics as features to build VPM based on deep neural networks. Given that code metrics can perfectly quantify code characteristics such as size, coupling and complexity, and these characteristics are correlated to vulnerabilities, code metrics may represent valuable data that allow DNNs to learn the characteristics of vulnerable code.

To detect vulnerabilities, the source code of the source entity (file, class, function) is decomposed into reduced portions of code (few lines of code: slices) that are related to a specific type of vulnerability. In this study, only vulnerabilities that are related to pointer usage (PU) are considered. Then, the code metrics of each slice are calculated and used as input of the DNN which classify the slice as vulnerable or clean.

TABLE 2. Descriptive statistics about the original dataset.

	Number of vulnerable CGs	Number of clean CGs	Total
Part1: Library/API Function Call (FC)	13603	50800	64403
Part2: Array Usage (AU)	10926	31303	42229
Part3: Pointer Usage (PU)	28391	263450	291841
Part4: Arithmetic	3475	18679	22154

C. DATA PREPARATION

In this sub-section, we present the data preparation. We begin by presenting the original dataset (slices dataset), after that, we present the proposed dataset (code metrics dataset) which is generated from the original dataset and used to train and validate the proposed VPMs.

1) ORIGINAL DATASET (SLICES DATASET)

The data used to carry out the experiments are generated from a public dataset [29] recently proposed by [18]. This dataset contains 420,627 labelled code slices (CSs), including 56,395 vulnerable CSs and 364,232 clean CSs. It is organized into four parts. Each part contains CSs about a specific type of vulnerability. The actual version of this dataset considers the following types of vulnerabilities: Library/API Function Call (vulnerabilities that are related to library/API function calls), Array Usage (vulnerabilities that are related to improper use of arrays), Pointer Usage (vulnerabilities that are related to improper use of pointers such as use after free vulnerability), Arithmetic Expression (vulnerabilities that are related to improper use of arithmetic expressions such as integer overflow vulnerability). The size (number of CSs) of each part is shown in Table 2. Four examples of labeled CSs taken from the original dataset are shown in Figure 2. The label of each CS is indicated in the last line, "1" means that the CS is vulnerable and "0" means that the CS is clean.

2) CODE METRICS DATASET

We prepared a dataset of 18 code metrics (the features) calculated from the labeled CSs of the original dataset. For the sake of minimizing the number of experiments, only the PU part (the biggest one) of the original dataset is considered.

CSs are composed of a few lines of code related to vulnerabilities (figure 3). Therefore, we calculated only Line of Code (LOC) metrics and code metrics related to the line and the token granularities like McCabe [30] and Halstead metrics [31]. Other types of code metrics related to other granularities such as class granularity (Object-Oriented Metrics) and function granularity (FanIn, FanOut, etc.) cannot be calculated. The actual version of the dataset contains the following metrics:

- LOC metrics (Physic lines: number of total lines, Empty lines: number of empty lines, Lines of comments: number of lines of comments, Lines of the program: number of program lines (directive, definition, declaration, commands...)).
- McCabe Metrics (McCabe number: The cyclomatic complexity)
- Halstead metrics (n1: number of distinct operators, n2: number of distinct operands, N1: total number of operators, N2: total number of operands, n: program vocabulary, N: program length, N': calculated program length, V: Halstead volume, D: difficulty, E: effort, T: the time required to program, B1: number of delivered Bugs 1, B2: number of delivered Bugs 2)

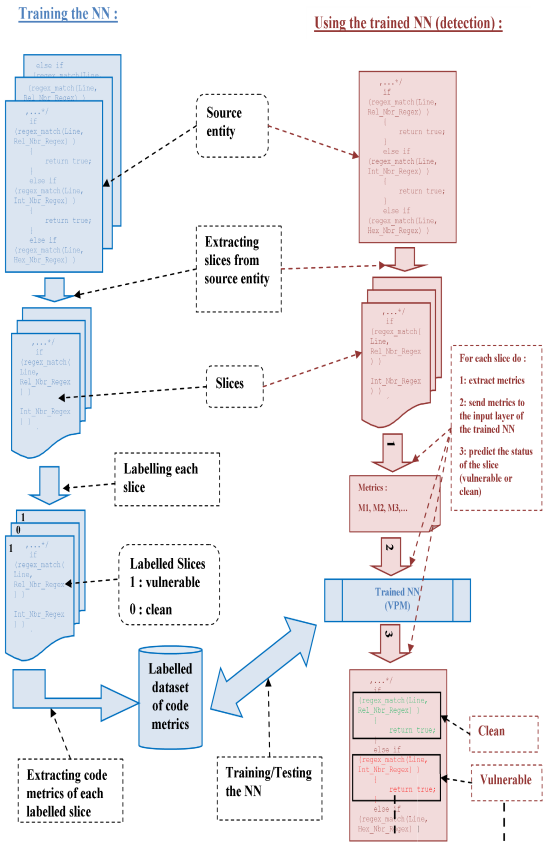

FIGURE 2. Proposed Approach.

The preparation of the dataset followed two steps. The first step consists of parsing the original dataset to separate each CS then calculate its code metrics. The second step consists of eliminating any redundant data. The final version of the dataset [32] which is used in experiments contains 95351 instances. To work with the Java API of Weka, the actual version of the dataset is in the ARFF (Attribute-Relation File Format) format.

ARFF file is an ASCII text file that describes a list of instances sharing a set of attributes. ARFF files have two distinct sections. The first section is the header information, which is followed by the data information. The header contains the name of the relation and a list of the attributes (the columns in the data). The data section of the file contains the data declaration line and the actual instance lines [33].

```
------------------------------
89053 199236/buffer_underrun_dynamic.c pbuf 173
void dynamic_buffer_underrun_009 ()
int * * pbuf [ 5 ] = { & buf1 , & buf2 , & buf3 , & buf4 , & buf5 } ;
int i , j = 4 ;
for(i=0;i<5;i++)
* ( ( * pbuf [ i ] ) + j ) = 5;

1
------------------------------
89054 199236/buffer_underrun_dynamic.c buf5 179
void dynamic_buffer_underrun_009 ()
int * buf5 = ( int * ) calloc ( 5 , sizeof ( int ) ) ;
free ( buf5 );

0
------------------------------
89055 199236/buffer_underrun_dynamic.c buf4 178
void dynamic_buffer_underrun_009 ()
int * buf4 = ( int * ) calloc ( 5 , sizeof ( int ) ) ;
free ( buf4 );

0
------------------------------
89056 199236/buffer_underrun_dynamic.c buf3 177
void dynamic_buffer_underrun_009 ()
int * buf3 = ( int * ) calloc ( 5 , sizeof ( int ) ) ;
free ( buf3 );

0
------------------------------
```

FIGURE 3. Examples of labeled CSs from the original dataset.

3) BALANCING THE DATASET

For most software projects, the percentage of vulnerable source lines is very low and the percentage of clean source lines is very high. As a consequence, the extracted data may inherit this characteristic which leads to an imbalanced dataset (the number of negative instances is very higher than the number of positive instances).

Training a VPM from such an imbalanced dataset is often challenging because the VPM may be biased towards the major class (negatives) and hence it only learns to predict everything as negatives and ignores the minor class (positives). Therefore, undersampling is a technique that is often used to balance the training set [11], [13], [15]. With this technique, all the positive cases in the training set are retained, while only a subset of the negatives is selected. The sample of negatives is randomly chosen such that the number of positives matches the number of negatives. The result is a perfectly balanced training set, which is used to build the VPM. The testing set is never altered, to preserve the correct testing conditions (the number of vulnerable source lines is very low and the percentage of clean source lines is very high). To balance our data, we used undersampling in the experiments using the implementation provided by the Weka API, the SpreadSubsample unsupervised filter [33].

D. DNN MODEL

1) DNN MODEL SELECTION

The prior studies that investigated vulnerability detection and prediction [17], [18], [24] have used automatically-learned features as input for DNN. They used the technique of word embedding to learn features which is a technique inspired by the previous applications of DNN in the field of natural language processing. Therefore, they used DNN models (Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNN)) which are suitable for automatically-learned features and which are commonly used in that field.

In this study, we aim to investigate whether software metrics are accurate when used as features for deep learning to detect vulnerabilities. The nature of the data (code metrics) used in this study is different from the data used in previous studies. Therefore, we prefer to use the DNN model that is suitable for tabular data. The Multi-Layer Perceptron (MLP) is a widely used DNN and it is very suitable for tabular data such as the code metrics dataset prepared in this study. Therefore, we used the MLP model to carry out the experiments and to draw the final conclusion.

Nevertheless and for the sake of completeness, we used also the LSTM (Long Short-Term Memory) which is a variant of the RNN model.

2) DNN MODEL CONSTRUCTION

A typical DNN has one input layer, one or more hidden layers, and one output layer. In this study, we investigated several architectures with different numbers of hidden layers and different neuron numbers in each hidden layer.

To construct, train and test the DNN models used to predict vulnerable CSs, we used the Java API of the WekaDeeplearning4j [34]. WekaDeeplearning4j is a Weka package based on the Deeplearning4j library [35]. The WekaDeeplearning4j package supports fully connected feedforward networks, convolutional networks, and recurrent networks. It also provides data loaders for standard tabular data, as well as image, text, and sequence data.

IV. EXPERIMENTS

To evaluate the performances of the DNN model, we used a standard k-folds cross-validation technique. In this technique, the instances of the dataset are randomly divided into $\boldsymbol{k}$ folds of equal size. Iteratively, each fold is retained as the testing set. That way, the VPM is trained with the samples in the other k-1 folds (training set) and used to predict the class of the CSs in the testing set. To reduce the computation time, we carried out the experiments using $k = 3$. In large datasets such as our dataset, 3-folds cross-validation is quite accurate.

The prediction result of the model can be one of the four cases: True Positive (if a vulnerable CS is predicted as vulnerable by the model), True Negative (if a clean CS is predicted as clean by the model), False Positive (if a clean CS is predicted as vulnerable by the model), False Negative (if a vulnerable CS is predicted as clean by the model). From the number of TP, TN, FP, and FN several performance indicators can be calculated.

We used four well-known and widely used performance indicators: **recall, precision, false-positive rate** and **false-negative rate.** A perfect VPM must not miss any vulnerability (false negative $\approx 0\%$) and help to minimize maintenance

costs by predicting as vulnerable only the source entities that are actually vulnerable (false positive $\approx 0\%$). It must also make a correct and precise prediction (recall $\approx 100\%$ and precision $\approx 100\%$). In the following, we summarize the definitions and formulas of the used performance indicators:

- Recall: This indicator gives the percentage of vulnerable CSs that are correctly classified by the model.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} * 100 \quad (1)$$

- Precision: This indicator gives the percentage of CSs classified as vulnerable by the model and are actually vulnerable.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} * 100 \quad (2)$$

- FP Rate: This indicator measures the percentage of misclassification positive among the real negatives.

$$\text{FP Rate} = \frac{\text{FP}}{\text{FP} + \text{TN}} * 100 \quad (3)$$

- FN Rate: This indicator measures the percentage of negatives that are falsely classified as real positives.

$$\text{FN Rate} = \frac{\text{FN}}{\text{FN} + \text{TP}} * 100 \quad (4)$$

Optimizing the parameters of a DNN is very challenging, especially when working with a new type of data or when trying to solve a new problem that has not been solved before by DL. The hyper-parameters for the back-propagation algorithm are as follows: the learning rate is 0.01, the momentum is 0.01. The other main parameters are as follows: the batch size is 128, the number of epochs to train through is 2500. These values were chosen based on what experts in the field of DL recommend. We investigated the impact of using different architectures (different numbers of hidden layers and different numbers of neurons in each hidden layer). To combat over-fitting, we used early stopping implementation to stop training after 20 epochs without loss improvement on a separate validation set.

Because the metrics used to train the DNNs are continuous variables and to combat any negative influences such as overfitting, we considered normalizing the data in experiments.

The machine running experiments has a 4 GB of RAM and a CPU Intel Xeon E5-2650 V3 @ 2.30GHz 2.30GHz.

V. RESULTS AND DISCUSSION

Table 3 summarizes the obtained results using several MLP configurations and 3 folds cross-validation. Four performance indicators were considered to draw conclusions (Recall and Precision: the higher the better, while the FP rate and FN rate: the lower the better). "1 HL (6)" means DNN with one hidden layer that contains 6 neurons, "3 HL (6-3-2)" means three hidden layers with 6 neurons in the first HL, 3 neurons in the second HL and 2 neurons in the third HL.

As shown in Table 3 and Table 4, the DNN model gets very good vulnerability detection performances in terms of all performance indicators (precision: 74.0% - 76.9 % and recall: 73.4% - 76.6%). Obtained values in terms of FP Rate and FN Rate are slightly higher (23.36% - 26.56%) but they still in the range of acceptable values. We observed that increasing the complexity of the DNN (from "1 HL (6)" to "3HL (24-12-8)") improved the detection performances. But when the complexity of the DNN became high (over 32 neurons in each HL), the performances began to decrease. That's let to conclude that using simple architectures would be enough. Based on the obtained results, we can conclude that code metrics are very useful as training data to build a vulnerability detection system based on DL.

TABLE 3. Results using the balanced dataset.

Hidden layers	Recall (%)	Precision (%)	FP Rate (%)	FN Rate (%)
1 HL (6)	73.4	74.0	26.56	26.56
3 HL (6-3-2)	74.3	74.4	25.74	25.74
3 HL (12-6-4)	75.7	75.9	24.25	24.25
3 HL (24-12-8)	76.6	76.9	23.36	23.36
3 HL (32-32-32)	74.6	75.2	25.40	25.40
3 HL (64-64-64)	73.9	74.4	26.14	26.14

TABLE 4. Results in terms of additional performance indicators.

Hidden layers	F-Measure	Area Under ROC
1 HL (6)	0.7327	0.8027
3 HL (6-3-2)	0.7421	0.8031
3 HL (12-6-4)	0.7570	0.8300
3 HL (24-12-8)	0.7657	0.8371
3 HL (32-32-32)	0.7444	0.8210
3 HL (64-64-64)	0.7371	0.8095

TABLE 5. Results using LSTM.

Hidden layers	Recall (%)	Precision (%)	FP Rate (%)	FN Rate (%)	F-Measure	Area Under ROC
1 LSTM HL (3)	69.8	71.1	30.2	30.2	0.694	0.758
3 LSTM HL (6-3-2)	72.6	72.6	27.4	27.4	0.726	0.778
3 LSTM HL (12-6-4)	72.7	73.8	27.3	27.3	0.724	0.779
3 LSTM HL (24-12-8)	73.0	73.4	16.94	16.94	0.729	0.794

For the sake of completeness, we report in Table 4 additional performance indicators which are often reported in related work (F-Measure and Area Under ROC: the higher the better). And in Table 5 the obtained results using several LSTM configurations.

MLP's performances in terms of F-Measure are good: between 0.7327 and 0.7657 and in terms of Area Under ROC are very good: between 0.8027 and 0.8371. Also, MLP performances in terms of all performance indicators except the FP Rate and FN Rate were better than what we got using the LSTM. This because the LSTM is more effective in coping with sequential data involving context than with tabular data [36]. The LSTM get good values in terms of FP Rate and FN Rate.

To compare the proposed approach (which is based on using code metrics as training data) with previous approaches that used the same level of granularity (slice)

TABLE 6. Results using classic ML techniques.

Classifiers	Balanced Dataset	Recall (%)	Precision (%)	FP Rate (%)	FN Rate (%)
KNN	Yes	81.7	81.8	18.34	18.34
KNN	No	92.9	92.9	31.50	7.05
RF	Yes	83.1	83.1	16.94	16.94
RF	No	93.7	93.2	41.08	6.25

but automatically-learned features as training data [17], [18], we used only the reported results (Table 1: line 1 and line 2) because the data used in those studies are not available which made replicating them very difficult. Because the reported results are better than what we obtained in this study, we can also conclude that the code metrics are good but not the better data to use for building a vulnerability detection system based on DL.

We considered also comparing the vulnerability detection power of code metrics when used with DL and when used with classic ML. To do so, we used the same dataset to build and validate VPMs based on two well-known and widely used ML algorithms: random forest (RF) and k-nearest neighbor (KNN). The obtained results are reported in Table 6.

As can be seen, the classifiers built using code metrics and the two classic ML algorithms get excellent performance in terms of all performance indicators. Indeed, RF reached over than 93% in terms of precision and recall. And fewer than 17% in terms of FP rate and FN rate. KNN's performances are very close to RF's performances: 92.9% in terms of precision and recall and 18.34% in terms of FP Rate and FN Rate. These results let us conclude that code metrics are more suitable for ML than DL.

VI. LIMITATIONS

To get relevant and credible results, we took into accounts the following things during the study:

- Using a large dataset suitable for deep learning.
- Using cross-validation when evaluating the VPMs.
- Balancing the dataset to avoid any possible influence related to unbalanced data on results.
- Using the Java API of Weka and WekaDeeplearning4j which is a well-known and widely used tool in the fields of Data mining and ML. This avoids any problem related to miss implementation of the DNN and classifiers algorithms.
- Using a DNN model (MLP) suitable for the used data (code metrics which is a tabular data).

Nevertheless, the study presents the following limitations which must be addressed in the future works:

- The used data are generated from a large base of c/c++ open source codes. However, we cannot say whether the conclusions generalize outside of all types of software which is written in other programming languages. Therefore, the proposed approach must be evaluated for other types of software (web application and mobile applications) and for software written in other programming languages than c/c++. This represents an interesting open research problem for future works.
- The approach is evaluated only for one type of vulnerabilities.
- To optimize the DNN architecture (number of hidden layers, number of neurons in each hidden layer and other hyper-parameters related to the learning algorithm) we evaluated several architectures and took decisions based on recommendations of experts in the fields of DL. However other optimization techniques such as genetic algorithms must be used in future works.

VII. CONCLUSION

This study aims to evaluating code metrics as features for deep learning to detect software vulnerabilities. We proposed an approach to use code metrics in a finer granularity (slice: which contains few lines of code related to vulnerabilities). This can lead to discovering the exact location of vulnerable code. The finding shows that code metrics are good data that let DNN learn more about the characteristics of vulnerable code.

Other recent studies used techniques inspired by other applications of DNN such as automatic language processing to extract features and build vulnerability detection systems. When we compared the obtained results with the reported results of those studies, it's concluded also that despite the interesting results obtained, code metrics are good but not the better. The results showed also that code metrics can give excellent detection performance when used with classic ML techniques than when used with DL techniques.

As part of the study, we have proposed and made publicly available a dataset of code metrics that can be used by other researchers to evaluate and develop VPMs. Indeed, we plan to develop a real vulnerabilities detection system based on the proposed approach and using the classic ML techniques. Other interesting future work would be to address the limitations and open problems presented in the previous section.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their insightful comments and suggestions that helped us improve the article.

REFERENCES

[1] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Netw.*, vol. 61, pp. 85–117, Jan. 2015.
[2] Q. V. Le, "Building high-level features using large scale unsupervised learning," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process.*, May 2013, pp. 8595–8598.
[3] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Trans. Softw. Eng.*, vol. 22, no. 10, pp. 751–761, 1996.
[4] K. Gao, T. M. Khoshgoftaar, H. Wang, and N. Seliya, "Choosing software metrics for defect prediction: An investigation on feature selection techniques," *Softw., Pract. Exper.*, vol. 41, no. 5, pp. 579–606, Apr. 2011.

[5] 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. 2–13, Jan. 2007.
[6] M. M. M. Syeed, I. Hammouda, and T. Systä, "Prediction models and techniques for open source software projects," *Int. J. Open Source Softw. Processes*, vol. 5, no. 2, pp. 1–39, Apr. 2014.
[7] B. Turhan, "Software defect prediction modeling," in *Proc. 2nd Int. Symp. Empirical Softw. Eng. (IDoESE)*, 2007, pp. 90–95.
[8] P. Morrison, K. Herzig, B. Murphy, and L. Williams, "Challenges with applying vulnerability prediction models," in *Proc. Symp. Bootcamp Sci. Secur. (HotSoS)*, 2015, vol. 14, no. 2, pp. 1–9.
[9] M. Siavvas, E. Gelenbe, D. Kehagias, and D. Tzovaras, "Static analysis-based approaches for secure software development," in *Proc. Int. ISCIS Secur. Workshop*, 2018, pp. 142–157.
[10] I. Chowdhury and M. Zulkernine, "Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities?" in *Proc. ACM Symp. Appl. Comput. (SAC)*, 2010, p. 1963.
[11] 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, Nov. 2011.
[12] Y. Shin and L. Williams, "An empirical model to predict security vulnerabilities using code complexity metrics," in *Proc. 2nd ACM-IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, 2008, p. 315.
[13] 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, Jul. 2019.
[14] S. Moshtari and A. Sami, "Evaluating and comparing complexity, coupling and a new proposed set of coupling metrics in cross-project vulnerability prediction," in *Proc. 31st Annu. ACM Symp. Appl. Comput. (SAC)*, 2016, pp. 1415–1421.
[15] J. Walden, J. Stuckman, and R. Scandariato, "Predicting vulnerable components: Software metrics vs text mining," in *Proc. IEEE 25th Int. Symp. Softw. Rel. Eng.*, Nov. 2014, pp. 23–33.
[16] S. Zhang, D. Caragea, and X. Ou, "An empirical study on using the national vulnerability database to predict software vulnerabilities," in *Database and Expert Systems Applications* (Lecture Notes in Computer Science: Lecture Notes Artificial Intelligent: Lecture Notes Bioinformatics), vol. 6860. Berlin, Germany: Springer, 2011, pp. 217–231.
[17] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–15.
[18] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: A framework for using deep learning to detect software vulnerabilities," 2018, *arXiv:1807.06756*. [Online]. Available: https://arxiv.org/abs/1807.06756
[19] M. Weiser, "Program slicing," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 4, pp. 352–357, Jul. 1984.
[20] J. Silva, "A vocabulary of program slicing-based techniques," *ACM Comput. Surveys*, vol. 44, no. 3, pp. 1–41, Jun. 2012.
[21] K. Pan, S. Kim, and E. Whitehead, Jr., "Bug classification using program slicing metrics," in *Proc. 6th IEEE Int. Workshop Source Code Anal. Manipulation*, Sep. 2006, pp. 31–42.
[22] C. Catal, "Can we predict software vulnerability with deep neural network?" in *Proc. 19th Int. Multiconf. Inf. Soc. (IS)*, Oct. 2016, pp. 19–22.
[23] *Word2Vec*. Accessed: Aug. 12, 2019. [Online]. Available: http://radimrehurek.com/gensim/models/word2vec.html
[24] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *Proc. 17th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, Dec. 2018, pp. 757–762.
[25] C. Catal, A. Akbulut, E. Ekenoglu, and M. Alemdaroglu, "Development of a software vulnerability prediction Web service based on artificial neural networks," in *Proc. Pacific–Asia Conf. Knowl. Discovery Data Mining*, 2017, pp. 59–67.
[26] Y. Shin, "Exploring complexity metrics as indicators of software vulnerability," in *Proc. 3rd Int. Doctoral Symp. Empirical Softw. Eng.*, Kaiserslautern, Germany, 2008.
[27] M. Alenezi and I. Abunadi, "Evaluating software metrics as predictors of software vulnerabilities," *Int. J. Secur. Appl.*, vol. 9, no. 10, pp. 231–240, Oct. 2015.
[28] I. Abunadi and M. Alenezi, "Towards cross project vulnerability prediction in open source Web applications," in *Proc. Int. Conf. Eng. (MIS ICEMIS)*, 2015, pp. 1–5.
[29] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen. (2018). *SeVC and SyVC Dataset*. [Online]. Available: https://github.com/SySeVR/SySeVR/
[30] H. Watson, T. J. McCabe, and D. R. Wallace, *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*. Gaithersburg, MD, USA: NIST, 1996, pp. 1–114.
[31] V. Y. Shen, S. D. Conte, and H. E. Dunsmore, "Software science revisited: A critical analysis of the theory and its empirical support," *IEEE Trans. Softw. Eng.*, vols. SE–9, no. 2, pp. 155–165, Mar. 1983.
[32] M. Zagane and M. K. Abdi. (2019). *Code Metrics Dataset (PU)*. [Online]. Available: https://github.com/codemetricsdaset/slice_codemetricsdataset/
[33] Machine Learning Group at the University of Waikato. (2019). *Weka API Online Doc*. [Online]. Available: http://weka.sourceforge.net/doc.dev/
[34] S. Lang, F. Bravo-Marquez, C. Beckham, M. Hall, and E. Frank, "WekaDeeplearning4j: A deep learning package for weka based on Deeplearning4j," *Knowl.-Based Syst.*, vol. 178, pp. 48–50, Aug. 2019.
[35] (2019). *Deep Learning for Java*. [Online]. Available: https://deeplearning4j.org/
[36] 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, 2019.

MOHAMMED ZAGANE received the engineering degree in computer science from the University of Mascara, Mascara, Algeria, in 2007, and the magister degree in computer science from the Higher School of Computer Science, Algiers, Algeria, in 2010. He is currently pursuing the Ph.D. degree in computer science with Université Oran1, Oran, Algeria.

From 2009 to 2012, he was a Computer Engineer in the administration services of the state of Mascara, and since 2012, he has been an Assistant Professor with the Computer Science Department, University of Mascara. His research interests include the application of machine learning and deep learning in software engineering, software security, and image processing.

MUSTAPHA KAMEL ABDI received the master's and Ph.D. degrees in computer science from the Department of Computer Science, Université Oran1, Oran, Algeria. He is currently a Professor with the Department of Computer Science, and a Researcher with the RIIR Laboratory. His research interests include the application of artificial intelligence techniques to software engineering, software quality, software evolution, formal specification, systems analysis and simulations, data-mining, and information research.

MAMDOUH ALENEZI (Member, IEEE) received the M.S. degree from DePaul University, in 2011, and the Ph.D. degree from North Dakota State University, in 2014. He is currently the Dean of Educational Services, Prince Sultan University. He is also an Associate Professor of software engineering with the teaching emphasis on software engineering and software security. He has participates in organizing several international scientific conferences and editorial boards of the well-reputed journals. He has extensive experience in applying data mining and machine learning techniques to solve software engineering problems. He published more than 80 articles. He conducted several research areas and development of predictive models using machine learning to predict fault-prone classes, comprehend source code, and predict the appropriate developer to be assigned to a newly reported bug. His researches focus on software engineering, software security, machine learning, and open source software systems.