Received: 29 August 2018 | Revised: 18 October 2018 | Accepted: 25 November 2018
DOI: 10.1002/cpe.5103

WILEY

SPECIAL ISSUE PAPER

A performance evaluation of deep-learnt features for software vulnerability detection

Xinbo Ban | Shigang Liu | Chao Chen | Caslon Chua

School of Software and Electrical Engineering, Swinburne University of Technology, Victoria, Australia

Correspondence
Shigang Liu, John St, Hawthorn VIC 3122, Australia.
Email: shigangliu@swin.edu.au

Chao Chen, John St, Hawthorn VIC 3122, Australia.
Email: chaochen@swin.edu.au

Funding information
National Natural Science Foudation of China, Grant/Award Number: 61772405

Summary

Software vulnerability is a critical issue in the realm of cyber security. In terms of techniques, machine learning (ML) has been successfully used in many real-world problems such as software vulnerability detection, malware detection and function recognition, for high-quality feature representation learning. In this paper, we propose a performance evaluation study on ML based solutions for software vulnerability detection, conducting three experiments: machine learning-based techniques for software vulnerability detection based on the scenario of single type of vulnerability and multiple types of vulnerabilities per dataset; machine learning-based techniques for cross-project software vulnerability detection; and software vulnerability detection when facing the class imbalance problem with varying imbalance ratios. Experimental results show that it is possible to employ software vulnerability detection based on ML techniques. However, ML-based techniques suffer poor performance on both cross-project and class imbalance problem in software vulnerability detection.

KEYWORDS
deep learning, security, software vulnerability

1 | INTRODUCTION

Software vulnerability detection has long been an important issue in the community of software security. Today, programs are increasingly difficult to manage as they become more complex and diverse.[1,2] In a huge software system, any tiny vulnerability can cause the whole system to crash. Although the harmfulness and perniciousness of vulnerabilities are well known to all, it is impossible to create flawless software without any vulnerabilities because they seem to be inevitable.[1,3,4] Therefore, vulnerability detection plays an indispensable role in software development and maintenance.

In the field of cyber security, it has been a fundamental problem to mitigate and discovery software vulnerabilities. Many mitigation approaches to detect software vulnerabilities have been proposed by researchers, with most approaches demanding experts to locate where the software vulnerabilities are. This requires a lot of manual work and time. However, machine learning technique is proven to be useful in automatically identifying potential vulnerabilities, and it even plays an important role in cyber security.[5] It uses ground-truth for knowledge learning, then applies the learnt knowledge for new knowledge prediction. For example, Shin et al[2] applied models which are trained on old versions of Linux kernel and Firefox for vulnerabilities detection. Machine learning techniques were applied for detecting vulnerable components in Android applications by Scandariato et al.[6] Lin et al[7] used machine learning to learn rich features to solve the problem of potential compromise at the early stage of software development. Zhang et al[8,9] proposed a new Roubust statistical Traffic Classification (RTC) scheme using combination of supervised and unsupervised machine learning to face the challenge to the robustness of classification performance. In real-world scenarios, Chen et al[10] proposed a novel Lfun scheme that could greatly improve the accuracy of spam detection through using machine learning techniques.

In this paper, we propose a performance evaluation study of software vulnerability detection based on deep-learnt features. We automatically extract features from the source code using deep learning algorithm (ie, BiLSTM), which we call deep-learnt features. We also conducted three thorough experimental evaluation to analyze the performance of vulnerability detection, namely: 1) datasets with multiple and single vulnerability types; 2) cross-project software vulnerability detection; and 3) class imbalance problem in software vulnerability detection For the first experiment, we evaluate the deep-learnt features using six algorithms across five datasets. For the second experiment, we use one dataset for model building based on six classification algorithms, and another dataset for testing. For the third experiment, we set up five different imbalance ratios (IR) (ie, 20, 40, 60, 80, and 100) for each data set, where IR is the number of majority samples divided by number of minority samples. We then randomly create 10 datasets for each IR and perform 10 runs of 10 fold on each imbalanced datasets.

We organized the paper as follows. Related work is reviewed in Section 2. Section 3 presents our proposed deep learning scheme for feature representation learning. The experimental study and results are described in Section 4. Finally, Section 5 concludes this paper.

2 | RELATED WORK

Compared to the conventional vulnerability detection techniques which mainly focus on the characteristics of source code using code analysis approaches, machine learning techniques are specialized for recognition of patterns in data. Researchers have shown that machine learning techniques can be applied to vulnerability detection.

Morrison et al[11] pointed out that engineers prefer prediction at the code level though experiments because huge binaries inappropriately fitted for practical inspection. A scheme that was developed by Yamagnchi et al[12] could automatically deduce search patterns for taint-style vulnerabilities based on C source code. Alves et al[13] used a large and representative dataset to evaluate states of the art vulnerabilities prediction techniques.

Eschweiler et al[14] presented how important vulnerability detection is at the binary level, and proposed a new efficient approach used to search similar function at binary level. Grieco et al[15] uses machine learning approach to analyze memory corruption vulnerability in binary code using both static and dynamic lightweight features with Random Forest trained by dynamic features.

Prediction models which were trained by software metrics several times had been used to effectively predict fault.[16-18] As software metrics are valuable for understanding the code such as Cyclomatic Complexity and Number of lines of code (LoC) which received much attention as to which metric is the most predictive.[18]

A framework proposed by Chowdhury and Zulkernine[1] predicts vulnerabilities automatically using complexity, coupling, and cohesion (CCC) metrics and machine learning techniques. This approach was assessed with experiments on Mozilla Firefox using C4.5, Logistic Regression, Naive Bayes and Random Forests. It was shown that all classifiers could predict most vulnerable files in Mozilla Firefox. However, Shin and Williams[19] showed that vulnerability detection at component level performs badly, and Yamaguchi et al[20] proposed a finer-grained approach to extrapolate vulnerable functions.

Fontana et al[21] compared a few machine learning approaches to find out which ones could be more effective in the vulnerability detection. Using decision trees and random forest, higher than 96% accuracy were achieved in the detection of code smells.

Many papers compare the evaluation of different machine learning techniques. For example, approaches based on software metrics were compared with text mining techniques.[22] Three open source projects: Drupal, Moodle, and PhpMyAdmin were used in the experiments containing 223 vulnerabilities. Although the datasets clearly had representativeness problems, the recall of text mining models was higher than it of software metrics.

There are also studies on cross-project fault detection.[23] In Nam et al[24] and Pan et al,[25] transfer component analysis (TCA) is applied for prediction on files. TCA+ containing a normalisation selection process is an extension of TCA and significantly perform better during cross-project prediction tasks. The semantic features from Abstract Syntax Tree (ASTs) that come from Java source code was extracted by deep belief network (DBN) at file level.[26]

In addition, processes of extracting features primarily rely on experience, skills and heuristics of experts. The choice of extracted features affect not only detection capabilities but also probably on detection granularity. Perl et al[27] proposed an approach called VCCFinder that use Support Vector Machine (SVMs) to detect potential vulnerabilities. Scandariato et al[6] used bag-of-words (BoW) to represent features seeing a software component as terms with associated frequencies. Dam et al[28] presented a new approach based on deep learning could automatically predict vulnerabilities in source code. The approach captures the relationship of log context in the source code which contains much non-associated context between dependent code elements by Long Short-Term Memory (LSTM). Pinzger et al[29] firstly utilized metric that is derived from developer activities to infer the possibility of failures of post-release in software modules. Developer activity is linked to software security by Meneely and William.[30] They pointed out that outdated experience and potential bias were possibly carried by features that were chosen by knowledge domain individuals. Another disadvantage to this is that some manual features probably generalize not as well as expected which means the performance of features is bad for a project even if it is excellent in others.[31]

3 | DEEP FEATURE LEARNING

In this section, we present how to employ Bi-directional LSTM (BiLSTM) networks for feature representation learning at functional-level. We assume that there is a large amount of labeled data in the source domain; however, we are not able to apply classification algorithms on the source code directly. With deep learning in the field of natural language processing successfully used in many real-world scenarios such as

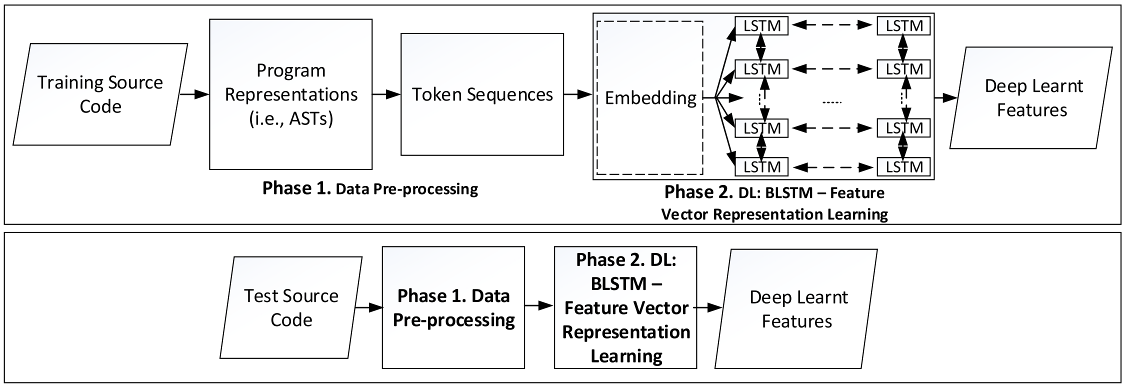
Training Source Code
Program Representations (i.e., ASTs)
Token Sequences
Embedding
LSTM
Deep Learnt Features
Phase 1. Data Pre-processing
Phase 2. DL: BLSTM – Feature Vector Representation Learning
Test Source Code
Phase 1. Data Pre-processing
Phase 2. DL: BLSTM – Feature Vector Representation Learning
Deep Learnt Features

FIGURE 1 Framework of the deep feature learning model

software vulnerability detection, and malware detection, we employ BiLSTM for deep feature learning. We hypothesize that deep learning could automatically learn useful feature representations that contain much richer information than the shallow features driven by domain knowledge.

At a high level, the deep feature learning process has two phases: data pre-processing and deep feature representation learning. As shown in Figure 1, we divided the vulnerability detection function in two phases. First, we preprocess program representations into token sequences. Second, we pre-train a deep feature model to map token sequences into dense vectors as their representations. It is worth noting that deep feature representation regarding the unclassified test cases can also be learnt by following the same processes, as hhown in the bottom box of Figure 1.

We choose ASTs as the program code representation for extracting features (which we call AST-based features). AST preserves programming patterns by depicting the structure and semantics of the code (ie, a function) in a tree structure. It also contains the relationships of components of a function and the function-level control flow. Therefore, we hypothesize that the vulnerable programming patterns can be unveiled by analysing functions' ASTs.

During the first, we obtain the source code of three open-source projects, and we parse ASTs from the code base for extracting AST-based features. In the second phase, we design an LSTM network for learning function-level AST representations using the extracted features as inputs. Both phases are described in detail as follows.

3.1 | Phase 1: data pre-processing

Parsing ASTs from source code is non-trivial since a build environment is needed for code compilation. We use "*CodeSensor*[*]" to extract ASTs from source code without supporting libraries and dependencies. "*CodeSensor* is a robust parser implemented in the work of Yamaguchi et al[32] based on the concept of *island grammars*.[33]

In order to obtain feature representations for building classification model, we convert parsed ASTs to vectors while preserving the structural information. Firstly, depth-first traversal is used by us to map ASTs nodes to vectors elements. The sequence of elements in a vector matters because they partially reflect the hierarchical position of the nodes in an AST. Therefore, the elements of vectors will be in a serialized form like: [*foo*, *int*, *params*, *int*, *x*, *stmts*, *decl*, *int*, *y*, =, *call*, *bar*,...], where *foo* is the name of a function, *int* denotes the return type of the function *foo*, *params* and *int* specify that function *foo* takes a parameter which is an *int* type, and so forth. For this textual vector, it is treated as a "sentence" with semantic meanings, and the elements of the vector and their sequence form the semantic meaning.

Subsequently, we blur the project-specific names because the projects used in this study are following distinct naming conventions. A universal naming criterion is required to eliminate project-specifics. Take function calls, for example, we simply use "call" to refer to a function call, ignoring the name of the called functions. Next, we need to tokenize textual elements of the AST nodes, where each textual element of vectors is linked to an integer by using a map. The "tokens" using integers uniquely identify each textual element. For instance, type "int" is mapped to "1," keyword "static" is mapped to "2," and so on. By doing this, each element within the vectors was replaced with numeric tokens with their sequence remains intact.

3.2 | Phase 2: deep feature representation learning

Using the works of Lin et al[34] and Li et al,[35] we employ BiLSTM for feature learning. Five layers are built into the architecture of our proposed LSTM-based network. The first layer is an embedding layer which maps each element of the sequence to a dense vector of fixed dimensionality. With embedding, discrete inputs, such as words, are converted to dense vectors which are non-sparse. More importantly, embedding can convert words to more meaningful vector representations for training an effective statistical model. The second layer is an LSTM layer which contains

[*] http://codeexploration.blogspot.com.au/

64 LSTM units in a bi-directional form (altogether 128 LSTM units). The LSTM units are capable of learning dependencies of each element for generating higher-level abstractions. The third layer is a global max pooling layer used for strengthening the signals. The last two layers are dense layers with "tanh" and "sigmoid" activation functions for converging a 128-dimensionality output to a single dimensionality. During the training process, the loss function that we attempt to minimize in our experiments is a binary cross entropy function.

Prior to acquiring the learned representations, we need to train the networks and evaluate whether the trained model is capable of offering acceptable detection performance. We use the extracted ASTs of both vulnerable and non-vulnerable functions from three open-source projects to feed the network. Subsequently, we feed the test data to the trained network to optimize the parameters of the trained model. Once the model is trained and the performance is satisfactory, we use the learned representations from the third layer of the networks as deep-learnt features. The learnt deep-learnt features is made up of 128-dimension vectors.

4 | PERFORMANCE EXPERIMENTS OF DEEP LEARNT FEATURES

4.1 | Datasets

Table 1 lists 5 datasets used in this paper. We collected three projects as our datasets that are all open-source: FFmpeg, LibPNG, and Asterisk in C/C++. The source code can be downloaded from the public code base or GitHub. For each dataset, we obtain the ground-truth by manually labelling the vulnerable functions recorded on two publicly available database sources, which are Common Vulnerability and Exposures (CVE) and the National Vulnerability Database (NVD) until 1st October 2017. After that, the remaining functions are selected by us as neutral ones excluding the identified vulnerable functions. Each project contains several types of vulnerabilities such as buffer errors, resource management errors, format string vulnerability, numeric errors, and divided by zero. In this paper, we focus on the function level. For vulnerabilities that span multiple functions or files, we discard them (ie, the vulnerable functions and the vulnerabilities which span across multiple functions and/or files are not included). We employ the model proposed in Section 3 for feature vector representation learning.

In conducting a comparative study, we adopt the datasets published by Li et al[35] We download 2 additional datasets and obtain the code gadgets as feature presentations according to what is described in the paper, where CWE119 contains buffer errors and CWE399 contains management errors.

4.2 | Experiments setup

Regarding the feature vector representation learning in terms of ASTs, we use the same setting as described by Lin et al.[34] We implement Bi-LSTM network on Keras (version 2.0.8) with TensorFlow (version 1.3.0) back-end. We use gensim package (version 3.0.1) for Word2vec embedding with all default settings. The computational system is a server running CentOS Linux 7 with two Physical Intel(R) Xeon(R) E5-2690 v3 2.60GHz CPUs and 96GB RAM. For code gadgets, we use the same parameter setting as Li et al.[35]

For the performane experiments, we employ six machine learning algorithms, which are C4.5 decision tree (J48), k-nearest neighbour(KNN), LDA, neural network(NN), random forest, and SVM. We use default setting in weka, and perform 10 runs of 10-fold cross-validation on each dataset. Specifically, for the experiments based on a single dataset, we report the averaged values of the total number of 100 experimental results. For class imbalance evaluation,[36] we perform a comparative study on a dataset with imbalance ratio (IR) of 20, 40, 60, 80, 100. For example, 20 means the number not vulnerable samples is 20 times of the number of vulnerable samples. For the cross-project experiment, we treat one dataset as training dataset while the other as a test dataset. For example, $A - B$ means A is the training dataset, B is the test dataset. we randomly select 80% samples from the training dataset for training the classification model and use the other project for the test. We repeat this 100 times and only report the averaged outcomes.

TABLE 1 Data information

Dataset/Project Name	# Vulnerable Samples	# Not Vulnerable Samples	# Total Samples	Feature Type
FFmpeg	191	4921	5112	AST
LibPNG	43	499	542	AST
Asterisk	56	14648	14704	AST
CWE119	10440	29313	39753	Code gadget
CWE399	7285	14600	21885	Code gadget

4.3 | Performance metrics

To evaluate the performance of classifiers in various environments, we adopted the metrics generally used in information retrieval.[37] We treat vulnerable functions as positive class and not vulnerable functions as a negative class in this study.

Positives (Vulnerable functions) and Negatives (Not vulnerable functions): Assume that there is a vulnerable sample *V* and the non-vulnerable (not vulnerable) class *NV*. The outputs of the classifiers are whether *V* belongs to *NV* or not. There is a very common method to evaluate the outputs is to make use of TP (true positive), FP (false positive), TN (true negative), and FN (false negative). The definitions of them are as follows:

- TP - vulnerable samples of class *V* correctly predicted as elements of *V*.
- FP - vulnerable samples not belonging to class *V* incorrectly predicted belonging to class *V*.
- TN - vulnerable samples belonging to class *NV* predicted as samples that belonging to class *NV*.
- FN - vulnerable samples of class *NV* incorrectly predicted belonging to class *NV*.

We report True Positive Rate(TPR) = Recall, False Positive Rate (FPR), Precision and F1-score, which can be defined as:

$$TPR = \frac{TP}{TP + FN} \tag{1}$$

$$FPR = \frac{FP}{FP + TN} \tag{2}$$

$$Precision = \frac{TP}{TP + FP} \tag{3}$$

$$F1 - score = \frac{2 * Precision * TPR}{Precision + TPR}. \tag{4}$$

4.4 | Experiment 1: single dataset with multiple and single types of vulnerabilities

We evaluate the performance of vulnerability detection performance of six classifiers across five different datasets which are generated using deep learnt features. Figure 2 shows True Positive Rate, False Positive Rate, Precision and F1 score of vulnerability detection of six machine

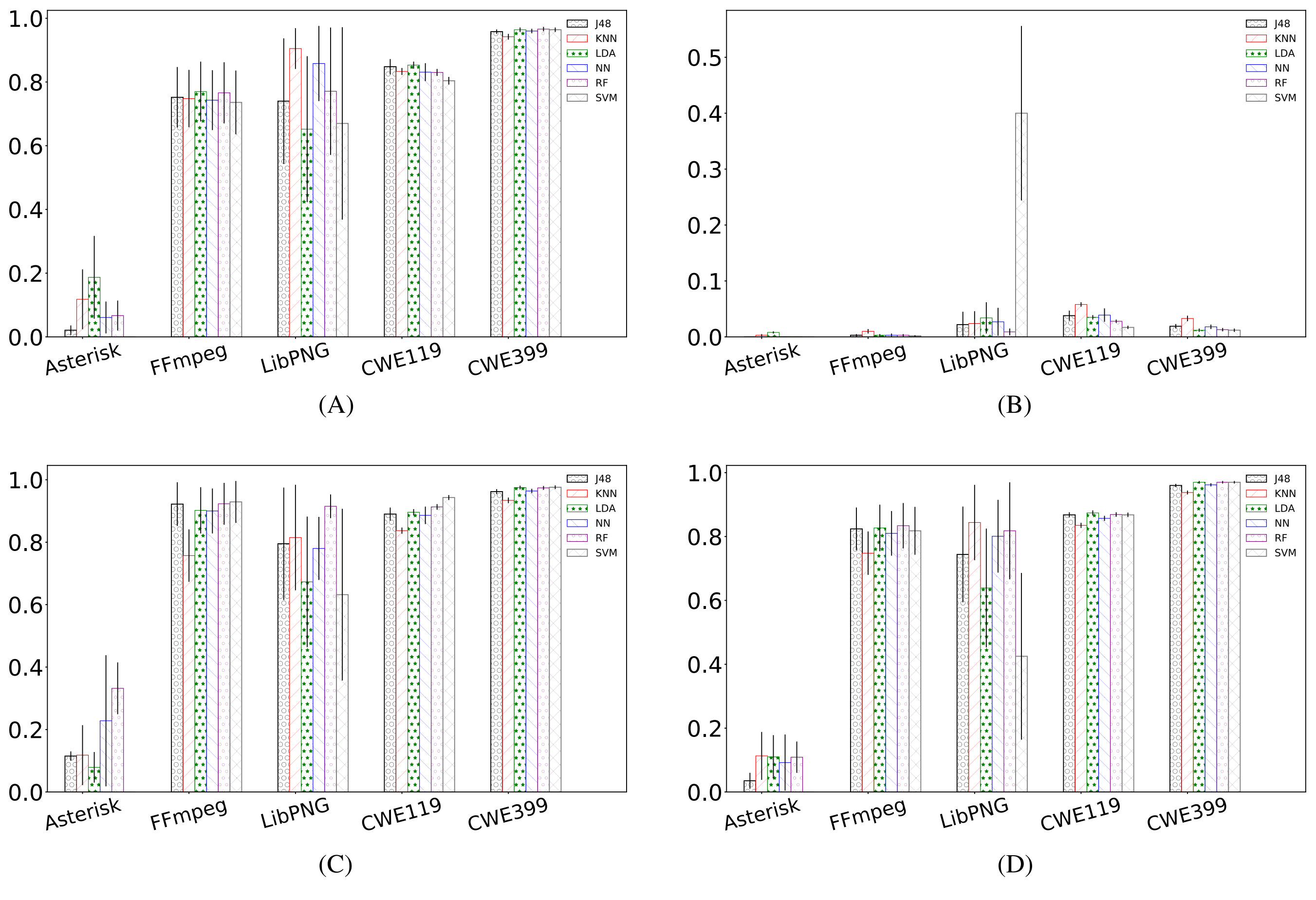

FIGURE 2 Experimental results based on single dataset/project in terms varies of classification algorithms. A, True Positive Rate; B, False Positive Rate; C, Precision; D, F1-score

learning classifiers on five different datasets. In terms of True Positive Rate, all six classifiers perform very well on three datasets, ie, FFmpeg, CWE119, and CWE399. Especially on CWE399, all classifiers even achieve over 90% TPR (see Figure 2A). However, TPR of the classifiers only ranges from 1% to 20% when evaluated on Asterisk dataset. There are two reasons for the low performance on these two datasets. Firstly, there are too few vulnerable samples in the datasets, only around 50 samples for each dataset. Secondly, the samples of vulnerable and non-vulnerable classes are too imbalanced in Asterisk dataset. We will discuss this class imbalance problem further in Section 4.6.

In Figure 2B, we can see that all classifiers on Asterisk, FFmpeg, and LibPNG have very low false positive rates except SVM, which are less than 1%. Classifiers' FPRs on CWE119 and CWE399 are slightly higher, ranging from 2% to 4%, which are also acceptable. Surprisingly, one can see that the FPR of SVM on LibPNG is more than 40%, which results in the low F1-score as well. This means SVM mis-classified many not vulnerable functions as vulnerable.

When looking at the precision in Figure 2C, we can find the same pattern as shown in Figure 2A: all the classifiers have very high precision (more than 90%, except *k*-NN on FFmpeg) on FFmpeg, CWE119 and CWE399, but low precision on Asterisk, due to the class imbalance issue in these two datasets.

TABLE 2 Experiments results of six machine learning techniques for cross-project software vulnerability detection

Training-Test	Classifier	TPR	FPR	Precision	F1 Score
Asterisk-FFmpeg	J48	0.001	0.001	0.053	0.003
	KNN	0.011	0.013	0.033	0.016
	LDA	0.404	0.044	0.278	**0.325**
	NN	0.042	0.002	0.380	0.074
	RF	0.000	0.000	NaN	NaN
	SVM	0.000	0.000	NaN	NaN
Asterisk-LibPNG	J48	0.000	0.020	0.000	NaN
	KNN	0.000	0.011	0.000	NaN
	LDA	0.047	0.175	0.024	**0.032**
	NN	0.000	0.002	0.000	NaN
	RF	0.000	0.002	0.000	NaN
	SVM	0.000	0.000	NaN	NaN
FFmpeg-Asterisk	J48	0.018	0.006	0.013	0.015
	KNN	0.002	0.010	0.001	0.001
	LDA	0.019	0.005	0.016	**0.017**
	NN	0.016	0.005	0.013	0.014
	RF	0.013	0.005	0.010	0.011
	SVM	0.004	0.005	0.003	0.003
FFmpeg-LibPNG	J48	0.000	0.007	0.000	NaN
	KNN	0.000	0.004	0.000	NaN
	LDA	0.000	0.000	NaN	NaN
	NN	0.000	0.004	0.000	NaN
	RF	0.000	0.000	NaN	NaN
	SVM	0.000	0.000	NaN	NaN
LibPNG-Asterisk	J48	0.071	0.029	0.010	0.018
	KNN	0.036	0.030	0.005	0.008
	LDA	0.232	0.425	0.002	0.004
	NN	0.054	0.002	0.081	**0.065**
	RF	0.018	0.014	0.005	0.008
	SVM	1.000	1.000	0.004	0.008
LibPNG-FFmpeg	J48	0.026	0.039	0.026	0.026
	KNN	0.005	0.063	0.003	0.004
	LDA	0.042	0.273	0.006	0.011
	NN	0.000	0.000	NaN	NaN
	RF	0.005	0.016	0.013	0.007
	SVM	1.000	1.000	0.039	**0.075**
CWE119-CWE399	J48	0.198	0.408	0.202	0.194
	KNN	0.500	0.450	0.357	0.417
	LDA	0.860	0.091	0.825	**0.842**
	NN	0.502	0.637	0.288	0.362
	RF	0.331	0.338	0.328	0.327
	SVM	0.009	0.027	0.181	0.017
CWE399-CWE119	J48	0.753	0.764	0.262	0.387
	KNN	0.596	0.250	0.460	0.519
	LDA	0.796	0.076	0.789	**0.792**
	NN	0.507	0.531	0.317	0.376
	RF	0.919	0.848	0.281	0.429
	SVM	0.039	0.014	0.494	0.071

The results show that classifiers' performance on F1-score will follow that of TPR and precision, as F1-score is a combined metric of TPR and precision. As shown in Figure 2D, F1-scores of all six classifiers on FFmpeg, CWE119 and CWE399 are no lower than 80% (except *k*-NN on FFmpeg). Particularly, the F1-scores of all six classifiers are more than 95% when evaluated on the CWE119 dataset. Again, the F1-scores are unacceptable (lower than 20% for all classifiers) when evaluated on Asterisk dataset.

4.5 | Experiment 2: Cross project software vulnerability detection

We evaluate the classifiers' performance in a cross project manner. That is to say, training and testing are performed on different datasets.

Table 2 shows the averaged results of TPR, FPR, Precision, and F1-score based on 8 scenarios. Since Asterisk, FFmpeg, and LibPNG datasets contain multiple types of vulnerabilities, we treat the experiments based on these three datasets as cross-project with multiple vulnerabilities. While CWE119 and CWE399 are datasets with a single type of vulnerability, we treat experiments based on these two datasets as single project with single vulnerability. We can see that machine learning-based techniques result in poor performance when facing cross-project software vulnerability detection based on multiple vulnerabilities per dataset. Take LibPNG-FFmpeg as an example, five out of six classifiers achieve TPR less than 0.05 and precision less than 0.02, which result in very low F1-score (less than 0.03). In this case, the classifiers are useless because it missed many vulnerabilities.

Moreover, since F1-score is the combined metric of TPR and precision, we pick up F1-score for experimental results explanation. We have reported that the performance of all classifiers on 8 different training-testing sets in this experiment. The bold number represents the best performance of the six algorithms tested for each set. As expected, the F1-scores of the classifiers are very low when training and testing are performed on different datasets, except for CWE119-CWE399 and CWE399-CWE119. The F1-score of these two scenarios is 84.2% and 79.2%, respectively. Specifically, we can see that LDA yields the best performance on CWE119-CWE399 and CWE399-CWE119 because it can identify most number of vulnerabilities, and it has the least number of mis-classified not vulnerable functions. One possible reason is that the deep-learnt features of CWE119 and CWE399 are linear spreadable, therefore, LDA which is a linear classifier performs the best.

The experimental results demonstrate that machine learning-based techniques are ineffective in cross-project software vulnerability detection with multiple vulnerabilities per project. However, it works well in the scenario of a single type of vulnerability per dataset. We suspect this is because the distribution divergence of the datasets with multiple types of vulnerabilities are more different than dataset with a single type of vulnerability. Another possible reason is that the deep-learnt features of FFmpeg, LibPNG, and Asterisk are based on ASTs, while the deep-learnt features of CWE119 and CWE399 are based on code gadgets, we doubt code gadgets based feature selection may be useful in helping the classifiers to identify more vulnerabilities.

4.6 | Experiment 3: imbalanced class problem in software vulnerability detection

We evaluate the impact of imbalance ratio (IR = non-vulnerable instances / vulnerable instances) of the same six machine learning algorithms on four datasets. We exclude Asterisk dataset because its IR is about 262, which is severe imbalance data. The experimental results using the Asterisk dataset is quite acceptable, thus we will not show the results based of this dataset. We only evaluate the class imbalance problem based on the other four datasets, which we think would be more useful in the real-world scenario. More specifically, we evaluate the classifiers' performance with IR ranging from 20 to 100.

Figure 3 to Figure 6 show the TPR, FPR, Precision, and F-measure of all classifiers on FFmpeg, CWE119, and CWE399. Overall, we can see that all metrics except FPR of the six classifiers become worse with the increase of IR. The reason why FPR becomes better is that more true negatives (non-vulnerable samples) are imported when IR is increasing. Among the six classifiers, LDA performs best while *k*-NN is worst. When taking a closer look at Figure 6D, which is showing the F1-score performance on dataset CWE399: the F1-score of LDA (the best performer) drops from 95% to 90%, when IR is ranging from 20 to 100, while F1-score of *k*-NN (the worst performer) from 82% to less than 70%. Similarly, the F1-scores of all other classifiers also decline with the increase of IR. With these results, we should take class imbalance issue into consideration when detecting software vulnerability. It is a major factor which can impact the detectors' performance.

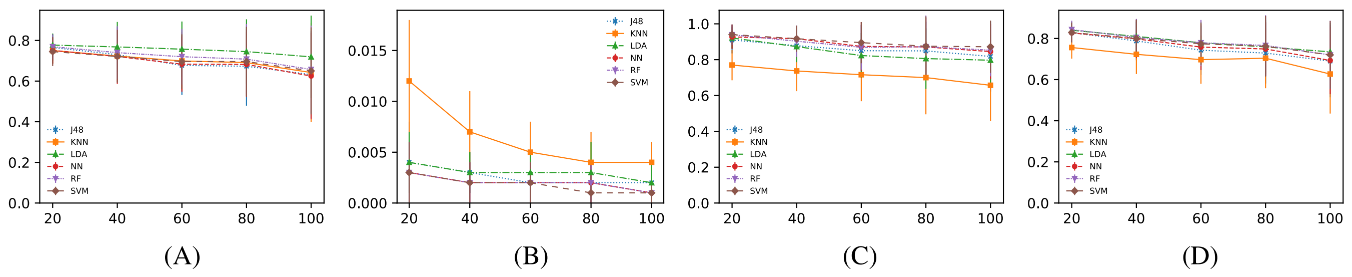

FIGURE 3 Experimental results based on FFmpeg dataset. A, True Positive Rate; B, False Positive Rate; C, Precision; D, F1 score

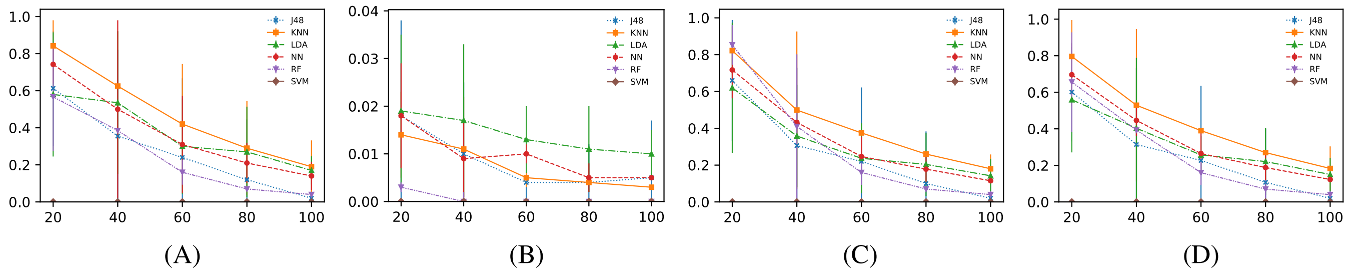

FIGURE 4 Experimental results based on LibPNG dataset. A, True Positive Rate; B, False Positive Rate; C, Precision; D, F1 score

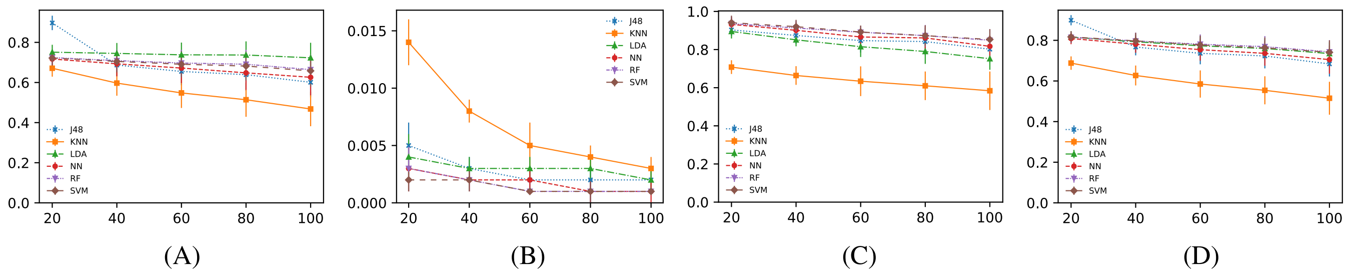

FIGURE 5 Experimental results based on CWE119 dataset. A, True Positive Rate; B, False Positive Rate; C, Precision; D, F1 score

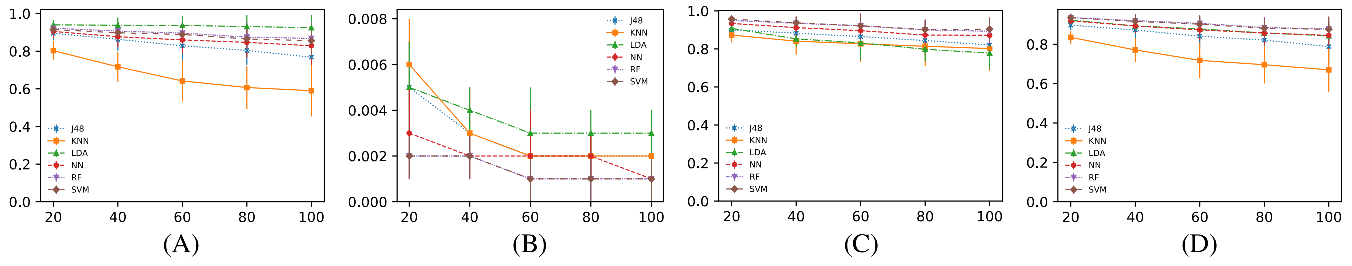

FIGURE 6 Experimental results based on CWE399 dataset. A, True Positive Rate; B, False Positive Rate; C, Precision; D, F1 score

5 | CONCLUSION

In this paper, we proposed a performance evaluation study of deep-learnt features for software vulnerability detection based on machine learning techniques. We employed deep learning (ie, BiLSTM) for feature vector representation learning. We also applied machine learning for the model building based on the deep-learnt features. We conducted three experiments: single dataset with multiple and single types of vulnerabilities; cross-project software vulnerability; and class imbalance problem on software vulnerability. Experimental results show that the machine learning-based model can be employed to identify potential vulnerabilities. However, machine learning-based techniques are suffering from poor performance when facing cross-project software vulnerability detection and class imbalance problem. We call on the community to develop cross-domain algorithms to deal with the cross-project problem specifically for software vulnerability detection. We also call on the researchers the community to pay attention to the class imbalance problem in the area of software vulnerability detection.

For future work, we plan to conduct deep learning-based cross-domain algorithms for software vulnerability detection in the scenario of cross-project problem. We also plan to conduct an evaluation study of software vulnerability detection on different kinds of techniques that developed for the class imbalance problem.

ORCID

Chao Chen https://orcid.org/0000-0003-1355-3870

REFERENCES

1. Chowdhury I, Zulkernine M. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *J Syst Archit*. 2011;57(3):294-313.
2. Shin Y, Meneely A, Williams L, Osborne JA. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Trans Softw Eng*. 2011;37(6):772-787.

3. Nguyen VH, Tran LMS. Predicting vulnerable software components with dependency graphs. In: Proceedings of the 6th International Workshop on Security Measurements and Metrics; 2010; Bolzano, Italy.
4. Vanegue J, Lahiri SK. Towards practical reactive security audit using extended static checkers. Paper presented at: 2013 IEEE Symposium on Security and Privacy; 2013; Berkeley, CA.
5. Liu L, De Vel O, Han QL, Zhang J, Xiang Y. Detecting and preventing cyber insider threats: a survey. *IEEE Commun Surv Tutor*. 2018;20(2):1397-1417.
6. Scandariato R, Walden J, Hovsepyan A, Joosen W. Predicting vulnerable software components via text mining. *IEEE Trans Softw Eng*. 2014;40(10):993-1006.
7. Lin G, Zhang J, Luo W, et al. Cross-project transfer representation learning for vulnerable function discovery. *IEEE Trans Ind Inform*. 2018;14(7):3289-3297.
8. Zhang J, Chen X, Xiang Y, Zhou W, Wu J. Robust network traffic classification. *IEEE/ACM Trans Netw*. 2015;23(4):1257-1270.
9. Zhang J, Xiang Y, Wang Y, Zhou W, Xiang Y, Guan Y. Network traffic classification using correlation information. *IEEE Trans Parallel Distributed Syst*. 2013;24(1):104-117.
10. Chen C, Wang Y, Zhang J, Xiang Y, Zhou W, Min G. Statistical features-based real-time detection of drifted Twitter spam. *IEEE Trans Inf Forensics Secur*. 2017;12(4):914-925.
11. Morrison P, Herzig K, Murphy B, Williams L. Challenges with applying vulnerability prediction models. In: Proceedings of the 2015 Symposium and Bootcamp on the Science of Security; 2015; Urbana, IL.
12. Yamaguchi F, Maier A, Gascon H, Rieck K. Automatic inference of search patterns for taint-style vulnerabilities. Paper presented at: 2015 IEEE Symposium on Security and Privacy; 2015; San Jose, CA.
13. Alves H, Fonseca B, Antunes N. Experimenting machine learning techniques to predict vulnerabilities. Paper presented at: 2016 Seventh Latin-American Symposium on Dependable Computing (LADC); 2016; Cali, Colombia.
14. Eschweiler S, Yakdan K, Gerhards-Padilla E. discovRE: efficient cross-architecture identification of bugs in binary code. Paper presented at: 23rd Annual Network and Distributed System Security Symposium (NDSS); 2016; San Diego, CA.
15. Grieco G, Grinblat GL, Uzal L, Rawat S, Feist J, Mounier L. Toward large-scale vulnerability discovery using machine learning. In: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy; 2016; New Orleans, LA.
16. Briand LC, Wüst J, Daly JW, Porter DV. Exploring the relationships between design measures and software quality in object-oriented systems. *J Syst Softw*. 2000;51(3):245-273.
17. Ostrand TJ, Weyuker EJ, Bell RM. Predicting the location and number of faults in large software systems. *IEEE Trans Softw Eng*. 2005;31(4):340-355.
18. Menzies T, Greenwald J, Frank A. Data mining static code attributes to learn defect predictors. *IEEE Trans Softw Eng*. 2007;33(1):2-13.
19. Shin Y, Williams L. Can traditional fault prediction models be used for vulnerability prediction? *Empir Softw Eng*. 2013;18(1):25-59.
20. Yamaguchi F, Lindner F, Rieck K. Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning. In: Proceedings of the 5th USENIX Conference on Offensive Technologies; 2011; San Francisco, CA .
21. Fontana FA, Mäntylä MV, Zanoni M, Marino A. Comparing and experimenting machine learning techniques for code smell detection. *Empir Softw Eng*. 2016;21(3):1143-1191.
22. Walden J, Stuckman J, Scandariato R. Predicting vulnerable components: software metrics vs text mining. Paper presented at: 2014 IEEE 25th International Symposium on Software Reliability Engineering; 2014; Naples, Italy.
23. Poon WN, Bennin KE, Huang J, Phannachitta P, Keung JW. Cross-project defect prediction using a credibility theory based naive Bayes classifier. Paper presented at: 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS); 2017; Prague, Czech Republic.
24. Nam J, Pan SJ, Kim S. Transfer defect learning. Paper presented at: 2013 35th International Conference on Software Engineering (ICSE); 2013; San Francisco, CA.
25. Pan SJ, Tsang IW, Kwok JT, Yang Q. Domain adaptation via transfer component analysis. *IEEE Trans Neural Netw*. 2011;22(2):199-210.
26. Wang S, Liu T, Tan L. Automatically learning semantic features for defect prediction. Paper presented at: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE); 2016; Austin, TX.
27. Perl H, Dechand S, Smith M, et al. VCCFinder: finding potential vulnerabilities in open-source projects to assist code audits. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security; 2015; Denver, CO.
28. Dam HK, Tran T, Pham T, Ng SW, Grundy J, Ghose A. Automatic feature learning for vulnerability prediction. 2017. arXiv preprint arXiv:1708.02368.
29. Pinzger M, Nagappan N, Murphy B. Can developer-module networks predict failures? In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering; 2008; Atlanta, Ga.
30. Meneely A, Williams L. Secure open source collaboration: an empirical study of Linus' law. In: Proceedings of the 16th ACM Conference on Computer and Communications Security; 2009; Chicago, IL.
31. Zimmermann T, Nagappan N, Gall H, Giger E, Murphy B. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering; 2009; Amsterdam, The Netherlands.
32. Yamaguchi F, Lottmann M, Rieck K. Generalized vulnerability extrapolation using abstract syntax trees. In: Proceedings of the 28th Annual Computer Security Applications Conference; 2012; Orlando, FL.
33. Moonen L. Generating robust parsers using island grammars. In: Proceedings of the Eighth Working Conference on Reverse Engineering; 2001; Stuttgart, Germany.
34. Lin G, Zhang J, Luo W, Pan L, Xiang Y. POSTER: vulnerability discovery with function representation learning from unlabeled projects. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS); 2017; Dallas, TX.
35. Li Z, Zou D, Xu S, et al. VulDeePecker: a deep learning-based system for vulnerability detection. 2018. arXiv preprint arXiv:1801.01681.

36. Liu S, Zhang J, Xiang Y, Zhou W. Fuzzy-based information decomposition for incomplete and imbalanced data learning. *IEEE Trans Fuzzy Syst*. 2017;25(6):1476-1490.

37. Liu S, Wang Y, Chen C, Xiang Y. An ensemble learning approach for addressing the class imbalance problem in Twitter spam detection. In: *Information Security and Privacy: 21st Australasian Conference, ACISP 2016, Melbourne, VIC, Australia, July 4-6, 2016, Proceedings, Part I*. Cham, Switzerland: Springer International Publishing; 2016.

How to cite this article: Ban X, Liu S, Chen C, Chua C. A performance evaluation of deep-learnt features for software vulnerability detection. *Concurrency Computat Pract Exper*. 2018;e5103. https://doi.org/10.1002/cpe.5103