



Deep Learning-Based Vulnerable Function Detection: A Benchmark

Guanjun Lin¹, Wei Xiao², Jun Zhang^{1(✉)}, and Yang Xiang¹

¹ School of Software and Electrical Engineering, Swinburne University of Technology, Hawthorn, Melbourne, VIC 3122, Australia

junzhang@swin.edu.au

² School of Computer Science and Engineering, Changchun University of Technology, Changchun, Jilin Province, China

Abstract. The application of Deep Learning (DL) technique for code analysis enables the rich and latent patterns within software code to be revealed, facilitating various downstream tasks such as the software defect and vulnerability detection. Many DL architectures have been applied for identifying vulnerable code segments in recent literature. However, the proposed studies were evaluated on self-constructed/-collected datasets. There is a lack of unified performance criteria, acting as a baseline for measuring the effectiveness of the proposed DL-based approaches. This paper proposes a benchmarking framework for building and testing DL-based vulnerability detectors, providing six built-in mainstream neural network models with three embedding solutions available for selection. The framework also offers easy-to-use APIs for integration of new network models and embedding methods. In addition, we constructed a real-world vulnerability ground truth dataset containing manually labelled 1,471 vulnerable functions and 1,320 vulnerable files from nine open-source software projects. With the proposed framework and the ground truth dataset, researchers can conveniently establish a vulnerability detection baseline system for comparison and evaluation. This paper also includes usage examples of the proposed framework, aiming to investigate the performance behaviours of mainstream neural network models and providing a reference for DL-based vulnerability detection at function-level.

Keywords: Vulnerability detection · Neural network · Function-level detection

1 Introduction

Deep Learning (DL), a breakthrough technique which has achieved promising results in many fields such as image processing and natural language processing (NLP), has also been widely applied for software code analysis [3] and for vulnerability detection [17, 18, 20, 21]. Various DL architectures, including the Multi-Layer Perceptron (MLP) [10, 29], the Convolutional Neural Network

(CNN) [11,16,27,32], and the Long-Short Term Memory (LSTM) [18,20,21] have been adopted for learning latent vulnerable code patterns from different software code representations (e.g., the Abstract Syntax Trees (ASTs) or the Control Flow Graphs (CFGs)). However, the aforementioned approaches were evaluated on self-constructed/-collected datasets, and/or compared with conventional code analysis methods. There is a lack of a unified benchmarking dataset for evaluating the effectiveness of these DL-based approaches and there is also the absence of a baseline system which can be easily replicated to act as a reliable performance metric for comparison and evaluation.

In this paper, we take a step forward to bridge this gap by proposing a benchmarking framework based on Keras [7] with TensorFlow [2] backend, providing one-click execution scripts for establishing a DL-based baseline system for vulnerability detection. The framework encapsulates six mainstream neural network models and can be easily extended to support different code embedding schemes and neural models. We also constructed a vulnerability dataset at two levels of granularity i.e., the file-level and the function-level. The dataset is labeled based on the information provided by the Common Vulnerabilities and Exposures (CVEs)¹ and the National Vulnerability Database (NVD)², which are publicly available vulnerability data repositories. With this dataset and the proposed framework, a DL-based baseline system for vulnerability detection can be conveniently established for performance comparison and evaluation. We have published the proposed framework and dataset at Github³. In summary, the contributions of this paper are two-fold:

- We developed a modularized benchmarking framework encapsulating six mainstream neural network models and two different code embedding schemes, providing one-click execution for building and testing vulnerability detection models. To guarantee the extendability, the framework offers APIs for easy integration of more neural network models and to support more code embedding solutions.
- We constructed a real-world vulnerability ground truth dataset for performance evaluation of vulnerability detection solutions. We manually checked nine open-source projects across 1,089 popular releases and labelled/collected 1,471 vulnerable and 59,297 non-vulnerable source code functions. We also record 1,320 vulnerable and 4,460 non-vulnerable files.

The rest of this paper is organized as follows: Sect. 2 reviews the existing studies which applied DL techniques for vulnerability detection. Section 3 details the design and implementation of the proposed framework. We also introduce our proposed dataset and the known datasets in this field. In Sect. 4, we provide case studies to demonstrate how the proposed framework facilitates the building of the baseline systems using the different datasets. Section 5 concludes the paper.

¹ <https://cve.mitre.org/>.

² <https://nvd.nist.gov/>.

³ <https://github.com/DanielLin1986/Function-level-Vulnerability-Detection>.

2 Related Work

The successes of neural techniques in many areas, particularly in the field of NLP, motivated researchers to apply neural networks for code analysis for the detection of software defects and vulnerabilities. Early researchers adopted fully connected networks (a.k.a the Deep Neural Networks (DNNs) or the MLP) for detecting vulnerabilities in PHP applications [29], Linux programs [10] and Android applications [9, 23]. Nevertheless, the approaches proposed by these studies are task-/project-specific. Thus, no performance comparison was made among these studies.

Later studies generally built on the assumption that software code contains semantics and syntactic resembling the natural languages. Therefore, ideas and techniques from the NLP field have been applied for learning code semantics indicative of software vulnerabilities. The CNN (e.g., the text-CNN [13]), which can learn high-level representations from small context windows, has been applied for detecting vulnerabilities at assembly level [16] and at source code function-level [11, 27]. Another line of studies applied variants of Recurrent Neural Network (RNN) (e.g., the bidirectional LSTM network) for learning vulnerable code patterns [17–21]. The authors assumed that the vulnerable code semantics could be revealed by analyzing a long-range code context which could be achieved by using the LSTM network.

Most recently, researchers proposed more expressive models by constructing complex network structures. Wu et al. [32] added convolutional layers on top of an LSTM network for identifying vulnerable Linux programs. Le et al. [15] built their model on a Maximal Divergence Sequential Auto-Encoder (MDSAE) for extracting representations from sequences of machine instructions. Choi et al. [6] and Sestili et al. [28] applied the memory network [30, 31] for detecting buffer overflow vulnerabilities. However, due to each study using self-constructed/-collected dataset, there was no systematic performance comparison conducted across different approaches to indicate their effectiveness.

3 Benchmarking Framework

In this section, we introduce the design of the proposed benchmarking framework and our proposed dataset which can be utilized for establishing a baseline system for vulnerability detection. We also suggest a new metric for evaluating the performance vulnerability detectors.

3.1 Architecture and Implementation

Fig. 1 illustrates the modularized implementation of the proposed framework. It consists of three modules: the code encoding/embedding module, the training module and test module. It is a common practice to convert text/code tokens to vector representations so that they are acceptable by the underlying Machine Learning (ML) algorithms. More importantly, we aim at preserving the text/code

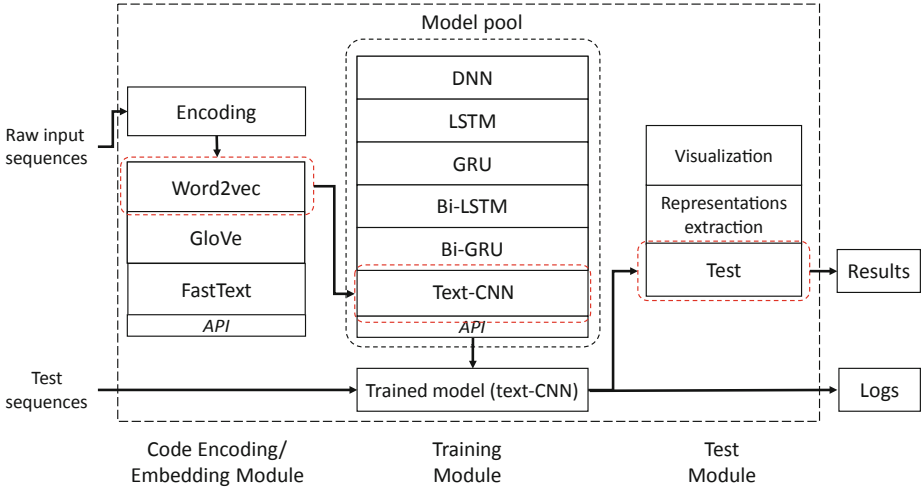


Fig. 1. The proposed benchmarking framework consists of three modules: the code encoding/embedding module, the training module and the test module. In the training phase, it allows users to choose different embedding schemes and different neural network models for building vulnerability detectors. In the test phase, it enables users to test the trained network model or to obtain representations from an arbitrary layer of a trained network. The framework provides APIs for easy integration of word/code embedding schemes and neural network models.

semantics while converting the text/code tokens to meaningful vector representations which we call the embeddings. The encoding/embedding module is built to serve this purpose. The module wraps mainstream word embedding schemes to enable the textual inputs i.e., the raw input code sequences to be converted to meaningful embeddings when we plot these embedding in a vector space, the semantically similar code tokens will be in close proximity in that vector space. This allows the neural network models to learn from a rich source. At this stage, the framework encapsulates three popular word embedding models: the Word2vec [22] model, the GloVe [24] model and FastText model [5].

In the training phase, the training module allows users to choose one of the built-in neural network models from the model pool for building a vulnerability detector. The framework provides six mainstream neural models: the DNN, the text-CNN and four RNN variants (i.e., the LSTM [12], the Gated Recurrent Unit (GRU) [14], and their bidirectional forms (the Bi-LSTM and the Bi-GRU)). During the test phase, users can feed the test data to a trained model and obtain detection results. The results are provided in a user-friendly format, including the confusion matrix and a CSV file recording the provability of each test sample containing the vulnerable code. Additionally, users can use trained neural networks as feature generators for generating neural representations from an arbitrary layer of a network. With this functionality, the generated representations can be used as features for downstream tasks (e.g., to train a random

forest classifier). The test module also provides Keras APIs for visualizing training/validation processes and supports TensorBoard⁴ logging system.

The framework provides one-click execution Python scripts, allowing users to invoke different modules of the framework to accomplish various tasks by specifying script arguments/parameters. For example, users can specify arguments such as *-train* or *-test* to switch the framework to training or test model. In addition, the script arguments allow users to select different built-in code embedding schemes and network models. A configuration file which contains plain text parameters is provided to offer more detailed options for model performance optimization. Users can either use the default settings for model training or customize the training process by fine-tuning the training settings and model hyperparameters.

We also provide easy-to-use APIs so that users can easily integrate new embedding schemes or implement their network models for training. The embedding API requires a Python dictionary object known as the *embedding index*. It is a table containing mappings between code tokens and the corresponding vector representations learned by the embedding method. The network model API accepts a Python class whose constructor takes one parameter which is the instance of the configuration file. Any models implemented using Keras or TensorFlow can be encapsulated in a Python class and invoked by the framework.

3.2 Dataset

The Proposed Dataset. The dataset we construct consists of nine popular open-source software projects written in C programming language, as listed in Table 1. It provides dual-granularity labelled samples, namely the vulnerable and non-vulnerable labels at function and file level. The vulnerable functions and files are labelled based on the description of the NVD and CVE web pages. In this paper, we focus on the vulnerabilities disclosed in the open-source projects because their source code is publicly available.

Typically, a vulnerability description on the NVD/CVE page specifies the exact location of the vulnerable code fragments in a particular version of a program. If the vulnerable code fragments are within a function boundary, we download the corresponding version of the source code of the program and label the source code function as vulnerable. Meanwhile, we label the file which contains the vulnerable function as vulnerable. A vulnerable file can contain at least one vulnerable function. For example, the vulnerable code fragments can span across multiple functions but they are within a file boundary, we only label the file as vulnerable. On the cases where the vulnerability description does not mention the location of vulnerable code fragments, we check the program’s Github and search the commit messages using the CVE ID as the keyword. We read through the commit message(s) of the returned result and identify the commit(s) that contain(s) the fix of the CVE. By analyzing the *diff* files, we identify the code fragment(s) associated with the CVE fix and the *diff* files allow us to track

⁴ https://www.tensorflow.org/guide/summaries_and_tensorboard.

Table 1. The number of vulnerable and non-vulnerable functions/files involved in the nine open-source projects in the proposed dataset.

Open-source projects	File-level		Function-level	
	# of non-vulnerable files collected	# of vulnerable files labeled	# of non-vulnerable functions collected	# of vulnerable functions labeled
Asterisk	862	84	17,755	94
FFmpeg	553	293	5,552	249
HTTPD	248	141	3,850	57
LibPNG	34	44	577	45
LibTIFF	94	151	731	123
OpenSSL	867	150	7,068	159
Pidgin	448	42	8,626	29
VLC Player	616	45	6,115	44
Xen	738	370	9,023	671
Total	4,460	1,320	59,297	1,471

the code prior to the fix. Then, we download the code before the fix and label them accordingly. By using this method, we can label some vulnerable files and functions which are not clearly described on the NVD and CVE pages. For the vulnerabilities which are not related to any functions or files (e.g., vulnerabilities caused due to the misconfiguration or incorrect settings). We simply discard these CVEs.

To collect the non-vulnerable files and functions, we download the latest release of the software projects at the time of writing. We assume that all the known vulnerability records in the CVE and NVD have been fixed in the latest release of a software project. To obtain the non-vulnerable files, we exclude the vulnerable files (despite these files have been fixed in the latest version) and use the remaining files as the non-vulnerable files. To obtain the non-vulnerable functions, we collect all the functions from the non-vulnerable files and label them as non-vulnerable.

The Synthetic Dataset. The synthetic vulnerability datasets provided by the Software Assurance Reference Dataset (SARD) project [1] contains artificially constructed test cases to simulate known vulnerable source code settings and patterns. The project consists of stand-alone test suits for C/C++ and Java, which are known as the Juliet Test Suites [4]. Each test site contains one main function so that the code can be compiled. In this paper, we collected all the C test cases from the SARD project and extracted 100,000+ functions from the test cases, forming a large synthetic function pool, as shown in Table 2.

The proposed dataset and the SARD project dataset form the base for benchmarking the proposed DL-based vulnerability detection framework. We aim to provide case studies of our framework and evaluate the performance behaviours of each neural network model on the proposed dataset containing real-world vulnerability samples and the SARD dataset having only synthetic samples.

Table 2. The number of vulnerable and non-vulnerable functions extracted from the SARD project.

Dataset	# of test cases	# of vulnerable C functions	# of non-vulnerable C functions
The SARD project	64,099	83,710	52,290

3.3 Performance Metrics

Precision, recall and F1-score are mainstream performance metrics for measuring the success of classification tasks. However, in the vulnerability detection scenario, one may face the severe data imbalance issue since there are significantly more non-vulnerable samples than the vulnerable ones in practice. For instance, the ratio of non-vulnerable functions to vulnerable ones is approximately 40 in our proposed dataset. Using metrics such as precision and recall would underestimate the detector’s performance because the classifier tends to fit the data distribution of the majority class and by default, it uses the 0.5 as the decision boundary in the cases of binary classification. Therefore, in this paper, we apply the top- k percentage precision ($P@K\%$) and top- k percentage recall ($R@K\%$) as the metrics for evaluating the performance of vulnerability detectors. Similar metrics are usually adopted in the context of information retrieval system such as search engines for measuring how many relevant documents are acquired in all the top- k retrieved documents [8]. We use these metrics in the vulnerability detection context to simulate a practical case where the number of functions to be retrieved for inspection accounted for a small proportion of total functions due to the constraints of time and resources.

In the vulnerability detection context, the top- k percentage refers to a list of retrieved functions accounted for $k\%$ of the total functions in the test set which are ordered by their probabilities of being vulnerable. The $P@K\%$ denotes the proportion of actual vulnerable functions identified by the detector in the top- $k\%$ retrieved function list. The $R@K\%$ refers to the proportion of actually found vulnerable functions which are in the top- $k\%$ returned function list. For measuring the vulnerable class, the $P@K\%$ and $R@K\%$ can be calculated using following equations:

$$P@K\% = \frac{TP@k\%}{TP@k\% + FP@k\%}, \quad R@K\% = \frac{TP@k\%}{TP@k\% + FN@k\%}, \quad (1)$$

where $TP@k\%$ is the true positive samples which are the actual vulnerable functions identified by the detector when retrieving $k\%$ most likely vulnerable functions. For example, there are 10,000 functions in a test set. After prediction, we examine 1% ($k = 1$) of the total functions which are the most likely to be vulnerable. That is, we retrieve top 100 functions ranked by their probability of being vulnerable and identify how many of these functions are actually vulnerable. Similarly, the $FP@k\%$ denotes the false vulnerable functions found by the

Table 3. The number of vulnerable and non-vulnerable functions partitioned on two datasets: the proposed dataset and the SARD dataset.

Dataset	Training set		Validation set		Test set	
	# of vul. functions	# of total functions	# of vul. functions	# of total functions	# of vul. functions	# of total functions
The proposed dataset	883	36,458	294	12,155	294	12,155
The SARD dataset	20,941	45,000	7,119	15,000	6,940	15,000

detector when returning $k\%$ most probable vulnerable functions. The $FN@k\%$ refers to the true vulnerable samples missed by the detector when returning $k\%$ functions.

4 Evaluation

This section evaluates the proposed benchmarking framework using the aforementioned real-world dataset and the SARD synthetic dataset for establishing baseline systems.

4.1 Experiment Settings and Environment

We set up two baseline systems using the proposed framework on two datasets. The first baseline system uses our constructed dataset, consisting of nine real-world open-source projects. It aims to investigate how the neural network models perform in a real-world scenario where severe data imbalance issue existed. The second baseline system uses the synthetic function samples from the SARD dataset. We extract functions from the test cases of the SARD project and randomly selected a subset of functions to form the dataset. The second baseline system aims to examine the behaviour of neural network models in an ideal scenario, so the dataset should not have data imbalance issue.

In this paper, we build the vulnerability detector at function-level and the Word2vec embedding scheme is chosen for embedding the code tokens. We use all the samples from our proposed dataset. For the SARD project, we randomly selected 35,000 vulnerable and 40,000 non-vulnerable C function samples from the test cases downloaded from the SARD data repository⁵. For both datasets, we partition the samples into the training, validation and test sets with the ratio of 6:2:2. The number of vulnerable and non-vulnerable samples in each set is listed in Table 3. For all the neural network models applied for case studies, we use the Stochastic Gradient Descent (SGD) optimizer with all default settings provided by Keras and the loss function to minimize is the binary cross-entropy.

⁵ <https://samate.nist.gov/SARD/testsuite.php#sardsuites>.

The neural models were implemented using *Keras* (version 2.2.4) [7] with a *TensorFlow* backend (version 1.13.1) [2]. The Word2Vec embedding software was provided by the *gensim* package (version 3.4.0) [26] using all default settings. The computational system used was a server running CentOS Linux 7 with two Physical Intel(R) Xeon(R) E5-2690 v3 2.60GHz CPUs and 256GB RAM with NVIDIA GTX 1080Ti GPUs.

4.2 Case Studies – The Bi-LSTM Network

In this case, we build the vulnerability detector using Bi-LSTM network and perform training and test on two datasets – the proposed dataset and the SARD dataset. We partition them into three sets according to Table 2. The Bi-LSTM network we design has seven layers. The first layer is the Word2vec embedding layer which converts the input code sequences to meaningful embedding vectors. The second and the third layers are bidirectional LSTM layers each of which contains 64 LSTM cells. A bidirectional layer contains a forward and a backward LSTM network so that the combined output can obtain information from both the preceding and succeeding context simultaneously. This allows the Bi-LSTM network to facilitate the learning of vulnerable code patterns which are associated with multiple lines of code [18, 21]. We concatenate the output of the LSTM networks of two directions and use a pooling layer for downsampling features. The last three layers of the network are dense layers, aiming to further converge the outputs to a single probability.

As the results are shown in Table 4, the Bi-LSTM network achieved better performance on the synthetic samples from the SARD dataset. When retrieving less than 50% of functions ranked by the probability of being vulnerable, the detector could identify all the vulnerable functions. When returning 50% of functions, all the vulnerable functions were found (represented by a 100% recall). In contrast, the Bi-LSTM network underperformed on the proposed dataset consisting of real-world function samples. When retrieving 1% of total functions which were considered being vulnerable, only 54% were actually vulnerable. However, when returning 20% of total functions, 87% of actual vulnerable functions could be identified and 99% of vulnerable function were found when retrieving only 50% of functions.

Table 4. The comparative results of the Bi-LSTM network on two datasets when retrieving different percentages of function samples ordered by their probabilities of being vulnerable.

Dataset	Precision and recall calculated when top $k\%$ functions were retrieved							
	1%		10%		20%		50%	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
The proposed dataset	54%	22%	18%	75%	10%	87%	5%	99%
The SARD dataset	100%	2%	100%	22%	100%	43%	93%	100%

4.3 Case Studies – The Text-CNN

Using the identical data partition setting mentioned in the previous case study, we build the vulnerability detector using the text-CNN implemented by Kim [13]. The only difference is that the convolution layer we use contains only 16 filters with four different sizes being 3, 4, 5 and 6, respectively. A filter can extract features from a small context window and various filters of different sizes are able to obtain different levels of features from the code sequences. This is different from the Bi-LSTM network which learns the long-range contextual dependencies from the code through the LSTM cells in the bidirectional structure. The filters of the text-CNN focus on extracting local features from small code contexts. Subsequently, the extracted features are passed to the pooling layers. After the pooling layers, the three dense layers are converted the features to a single probability.

Table 5. The comparative results of the text-CNN on two datasets when retrieving different percentages of function samples ordered by their probabilities of being vulnerable.

Dataset	Precision and recall calculated when top $k\%$ functions were retrieved							
	1%		10%		20%		50%	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
The proposed dataset	70%	29%	20%	81%	11%	90%	5%	97%
The SARD dataset	100%	2%	100%	22%	100%	43%	91%	98%

Table 5 shows the results of using text-CNN as the vulnerability detector. Similar to the results achieved by the Bi-LSTM network, the text-CNN performed well on the SARD dataset. The only difference between the Bi-LSTM network and the text-CNN on the SARD dataset is that when retrieving 50% of potentially vulnerable functions, the text-CNN could correctly identify 98% of actual vulnerable functions in the test set. Compared to the result achieved by the Bi-LSTM network (being 100%), the text-CNN underperformed. However, on our proposed dataset containing real-world samples, the text-CNN outperformed the Bi-LSTM network when returning less than 20% of the vulnerable functions. In particular, when retrieving 1% of vulnerable functions, the text-CNN could find 29% of total vulnerable functions. In contrast, the Bi-LSTM could identify only 22% of total vulnerable ones.

4.4 Case Studies – The DNN

Keeping the data partition setting unchanged, we build the vulnerability detector using the network containing fully connected layers i.e., the DNN. In contrast to the Bi-LSTM network and the text-CNN, the DNN is a generic structure not specifically designed for processing sequential data nor for spatial data. It is also “input structure-agnostic”. Namely, the network can take data of different

formats as inputs [25]. A DNN consists of multiple dense layers which map the inputs to space where data of different classes are more separable. In a sense, dense layers can be used to learn a non-linear function (with the non-linearity introduced by the activation functions) which better fits the complex and latent patterns of the data.

The DNN we use contains 6 layers. Identical to the Bi-LSTM network and the text-CNN, the first layer is the embedding layer for converting the code sequences to meaningful embeddings. The second layer flattens the outputs of the embedding layer so that the outputs can be 2-D tensors acceptable by the subsequent dense layers. The first dense layer contains 128 neurons. The number of the neurons in the second layer reduces to half and the same settings are applied for the third layer. The last layer has only one neuron which converges the outputs of the previous layer to a single probability.

Table 6. The comparative results of the DNN on two datasets when retrieving different percentages of function samples ordered by their probabilities of being vulnerable.

Dataset	Precision and recall calculated when top k% functions were retrieved							
	1%		10%		20%		50%	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
The proposed dataset	44%	18%	15%	62%	10%	80%	5%	96%
The SARD dataset	100%	2%	100%	22%	100%	43%	93%	100%

Table 6 lists the results of using DNN as the vulnerability detector on both the SARD and the proposed datasets. In contrast to the Bi-LSTM network and the text-CNN, the DNN underperformed on the proposed real-world dataset, achieved only 44% precision and 18% recall when retrieving 1% of the total functions which are most likely vulnerable. However, when retrieving 50% of the total functions, the performance of DNN was identical to that of the Bi-LSTM network and the text-CNN. On the SARD dataset, the DNN performed similarly compared with the other two networks.

4.5 Discussion

This section discusses the possible causes of the performance behaviours of the three network structures described in the aforementioned case studies. As shown in Tables 4, 5 and 6, when using the SARD dataset which consists of synthetic function samples, all the networks achieved similar and satisfactory performance. In contrast, the same networks underperformed on the proposed real-world dataset. The underlying reason is that the synthetic function samples are artificially constructed, following a template-like coding format. Therefore, the vulnerable and non-vulnerable code patterns can be easily learned and differentiated by the chosen neural networks. Whereas, the proposed dataset contains real-world function samples from open-source projects among which the code

structure and logic vary significantly. Thus, the vulnerable code patterns are diverse and hidden in the complex code logic, which are difficult to be captured by the neural network models.

When using the proposed real-world dataset, different network structures exhibited varying performance behaviours, demonstrating that network structures of different types have different capacities in terms of learning vulnerable code patterns. Compared with the Bi-LSTM network and the text-CNN, the DNN underperformed on the proposed real-world dataset. This indicated that the DNN which contains only the fully connected dense layers was less effective for learning the characteristics of the potentially vulnerable code. Nonetheless, the Bi-LSTM network and the text-CNN which are specifically designed for processing sequential and spatial data (i.e., the code sequences in our context) facilitated the learning of vulnerable code patterns, resulting in more accurate vulnerability detection on the real-world samples. The Bi-LSTM network which has bidirectional LSTM layers and the text-CNN which equips with multiple filters, are capable of handling the contextual dependencies among the elements in a sequence. Noticeably, the text-CNN achieved the best performance on the proposed dataset when retrieving less than 20% of the total functions. This revealed that the high-level features which were extracted from small context windows by the filters of the text-CNN contributed to more effective learning of vulnerable code patterns.

5 Conclusion and Future Work

In conclusion, we propose a DL-based framework, providing easy-to-use Python scripts for building/testing vulnerability detectors. To evaluate the usability of the framework and the performance of the built-in neural networks, we apply two datasets for a comprehensive benchmark. The first dataset is the SARD dataset containing synthetic vulnerability samples and the second one is a real-world vulnerability dataset which we manually constructed by labelling more than 1,300 vulnerable files and functions. We performed three case studies using the DNN, the Bi-LSTM network and the text-CNN network. The experiments showed that their performance behaviours were identical on the SARD synthetic dataset, indicating that the network structures were not an important variable affecting the performance on the synthetic vulnerability samples. Nevertheless, the performance behaviours of the three networks on the proposed real-world dataset revealed that the network models which were context-aware i.e., the text-CNN and the Bi-LSTM networks facilitated the detection of the real-world vulnerable samples.

The proposed real-world vulnerability dataset is still in a preliminary stage, requiring further effort to improve. Our future work will focus on collecting vulnerable and non-vulnerable code at binary-level, since many software tools are closed-source. Additionally, the current dataset does not include the patched vulnerabilities as the non-vulnerable samples. Being able to differentiate the vulnerabilities from their patched versions can be a key performance metric for

evaluating the effectiveness of the deep learning-based detectors. Thus, obtaining the patched vulnerable functions and files should also be our future work. Furthermore, we will continue to label more vulnerable samples and meanwhile, adding vulnerability type and severity information to the labeled vulnerabilities so that the dataset can be more useful to the research in this field.

References

1. Software assurance reference dataset project. <https://samate.nist.gov/SRD/> (2019). Accessed: 20 Aug 2019
2. Abadi, M., et al.: Tensorflow: a system for large-scale machine learning. In: OSDI, vol. 16, pp. 265–283 (2016)
3. Allamanis, M., Barr, E.T., Devanbu, P., Sutton, C.: A survey of machine learning for big code and naturalness. *ACM Comput. Surv. (CSUR)* **51**(4), 81 (2018)
4. Black, P.E., Black, P.E.: Juliet 1.3 Test Suite: Changes From 1.2. US Department of Commerce, National Institute of Standards and Technology (2018)
5. Bojanowski, P., Grave, E., Joulin, A., Mikolov, T.: Enriching word vectors with subword information. arXiv preprint [arXiv:1607.04606](https://arxiv.org/abs/1607.04606) (2016)
6. Choi, M.J., Jeong, S., Oh, H., Choo, J.: End-to-end prediction of buffer overruns from raw source code via neural memory networks. arXiv preprint [arXiv:1703.02458](https://arxiv.org/abs/1703.02458) (2017)
7. Chollet, F., et al.: Keras. <https://github.com/fchollet/keras> (2015)
8. Manning, C.D., Raghavan, P., Schütze, H.: Introduction to Information Retrieval. Cambridge University Press, Cambridge (2009)
9. Dong, F., Wang, J., Li, Q., Xu, G., Zhang, S.: Defect prediction in android binary executables using deep neural network. *Wireless Pers. Commun.* **102**(3), 2261–2285 (2018)
10. Grieco, G., Grinblat, G.L., 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, pp. 85–96. ACM (2016)
11. Harer, J.A., et al.: Automated software vulnerability detection with machine learning. arXiv preprint [arXiv:1803.04497](https://arxiv.org/abs/1803.04497) (2018)
12. Hochreiter, S., Schmidhuber, J.: Long short-term memory. *Neural Comput.* **9**(8), 1735–1780 (1997)
13. Kim, Y.: Convolutional neural networks for sentence classification. arXiv preprint [arXiv:1408.5882](https://arxiv.org/abs/1408.5882) (2014)
14. Kostadinov, S.: Understanding GRU networks, December 2017. <https://www.towardsdatascience.com>. Accessed 30 Apr 2019
15. Le, T., et al.: Maximal divergence sequential autoencoder for binary software vulnerability detection. In: Proceedings of the 7th International Conference on Learning Representations (2018)
16. Lee, Y.J., Choi, S.H., Kim, C., Lim, S.H., Park, K.W.: Learning binary code with deep learning to detect software weakness. In: KSII The 9th International Conference on Internet (ICONI) 2017 Symposium (2017)
17. Li, Z., et al.: SySeVR: A framework for using deep learning to detect software vulnerabilities. arXiv preprint [arXiv:1807.06756](https://arxiv.org/abs/1807.06756) (2018)
18. Li, Z., et al.: VulDeePecker: a deep learning-based system for vulnerability detection. In: Proceedings of NDSS (2018)

19. Lin, G., et al.: Software vulnerability discovery via learning multi-domain knowledge bases. *IEEE Transactions on Dependable and Secure Computing* (2019). <https://doi.org/10.1109/TDSC.2019.2954088>
20. 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 SIGSAC Conference on CCS*, pp. 2539–2541. ACM (2017)
21. Lin, G., et al.: Cross-project transfer representation learning for vulnerable function discovery. *IEEE Trans. Ind. Inform.* **14**(7), 3289–3297 (2018)
22. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. arXiv preprint [arXiv:1301.3781](https://arxiv.org/abs/1301.3781) (2013)
23. Peng, H., Mou, L., Li, G., Liu, Y., Zhang, L., Jin, Z.: Building program vector representations for deep learning. In: Zhang, S., Wirsing, M., Zhang, Z. (eds.) *KSEM 2015. LNCS (LNAI)*, vol. 9403, pp. 547–553. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25159-2_49
24. Pennington, J., Socher, R., Manning, C.: GloVe: global vectors for word representation. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1532–1543 (2014)
25. Ramsundar, B., Zadeh, R.B.: *TensorFlow for Deep Learning: From Linear Regression to Reinforcement Learning*. O'Reilly Media, Inc., Newton (2018)
26. Řehůřek, R., Sojka, P.: Software Framework for Topic Modelling with Large Corpora. In: *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks, ELRA, Valletta, Malta*, pp. 45–50, May 2010. <http://is.muni.cz/publication/884893/en>
27. Russell, R., et al.: Automated vulnerability detection in source code using deep representation learning. In: *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pp. 757–762. IEEE (2018)
28. Sestili, C.D., Snavely, W.S., VanHoudnos, N.M.: Towards security defect prediction with AI. arXiv preprint [arXiv:1808.09897](https://arxiv.org/abs/1808.09897) (2018)
29. Shar, L.K., Tan, H.B.K.: Predicting common web application vulnerabilities from input validation and sanitization code patterns. In: *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pp. 310–313. IEEE (2012)
30. Sukhbaatar, S., Weston, J., Fergus, R., et al.: End-to-end memory networks. In: *Advances in Neural Information Processing Systems*, pp. 2440–2448 (2015)
31. Weston, J., Chopra, S., Bordes, A.: Memory networks. arXiv preprint [arXiv:1410.3916](https://arxiv.org/abs/1410.3916) (2014)
32. Wu, F., Wang, J., Liu, J., Wang, W.: Vulnerability detection with deep learning. In: *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*, pp. 1298–1302. IEEE (2017)