




Deep neural-based vulnerability discovery demystified: data, model and performance

Guanjun Lin¹ · Wei Xiao² · Leo Yu Zhang³ · Shang Gao³ · Yonghang Tai⁴  · Jun Zhang⁵

Received: 20 July 2020 / Accepted: 25 March 2021 / Published online: 17 May 2021
© The Author(s), under exclusive licence to Springer-Verlag London Ltd., part of Springer Nature 2021

Abstract

Detecting source-code level vulnerabilities at the development phase is a cost-effective solution to prevent potential attacks from happening at the software deployment stage. Many machine learning, including deep learning-based solutions, have been proposed to aid the process of vulnerability discovery. However, these approaches were mainly evaluated on self-constructed/-collected datasets. It is difficult to evaluate the effectiveness of proposed approaches due to lacking a unified baseline dataset. To bridge this gap, we construct a function-level vulnerability dataset from scratch, providing in source-code-label pairs. To evaluate the constructed dataset, a function-level vulnerability detection framework is built to incorporate six mainstream neural network models as vulnerability detectors. We perform experiments to investigate the performance behaviors of the neural model-based detectors using source code as raw input with continuous Bag-of-Words neural embeddings. Empirical results reveal that the variants of recurrent neural networks and convolutional neural network perform well on our dataset, as the former is capable of handling contextual information and the latter learns features from small context windows. In terms of generalization ability, the fully connected network outperforms the other network architectures. The performance evaluation can serve as a reference benchmark for neural model-based vulnerability detection at function-level granularity. Our dataset can serve as ground truth for ML-based function-level vulnerability detection and a baseline for evaluating relevant approaches.

Keywords Vulnerability discovery · Deep learning · Function-level · Baseline dataset · Performance evaluation

Guanjun Lin and Wei Xiao have contributed equally to this work, and Yonghang Tai is the corresponding author.

✉ Yonghang Tai
taiyonghang@ynnu.edu.cn

Guanjun Lin
daniellin1986d@gmail.com

Wei Xiao
xiaowei@ccut.edu.cn

Leo Yu Zhang
leo.zhang@deakin.edu.au

Shang Gao
shang.gao@deakin.edu.au

Jun Zhang
junzhang@swin.edu.au

¹ School of Information Engineering, Sanming University, Sanming, Fujian Province, China

1 Introduction

Computer software is ubiquitous and affects all aspects of our lives daily. Vulnerabilities in the software might be exploited by attackers, thus leading to severe consequences

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

³ School of Information Technology, Deakin University, Geelong, VIC 3216, Australia

⁴ Yunnan Key Laboratory of Opto-electronic Information Technology, Yunnan Normal University, Kunming, Yunnan, China

⁵ School of Software and Electrical Engineering, Swinburne University of Technology, Melbourne, VIC 3122, Australia

such as data breaches, privacy leakage and financial loss. A recent data breach caused by a vulnerability in the Apache Struts has resulted in 143 million consumers' financial data to be compromised [1]. The global outbreak of WannaCry ransomware, caused by the exploitation of a vulnerability in the Server Message Block (SMB) protocol in early Windows, has affected millions of users. Consequently, it is essential to identify security vulnerabilities in software before they get exploited.

Vulnerability detection is a labor-intensive task, which requires expert knowledge [2]. To automate the detection process, many approaches have been proposed. In the field of software engineering, from the early rule-based methods [3] to the mainstream code analysis approaches such as symbolic execution [4], fuzzing test [5] and taint analysis [6], automated vulnerability discovery solutions have significantly improved the efficiency of code inspection for identifying vulnerabilities [7–10].

The data-driven approaches, powered by Machine Learning (ML) algorithms, have become alternative solutions for intelligent and effective vulnerability detection [11]. Researchers from software engineering and cybersecurity communities have proposed various ML-based approaches for vulnerability discovery. However, these approaches, including the ones utilizing deep learning techniques, were all based on self-collected/-constructed datasets for evaluating the effectiveness. For example, some studies chose various types of Mozilla software for empirical study [12, 13]. Others applied different open-source software projects for evaluation [7, 14–20]. Also, Android applications and the synthetically generated datasets were popular data sources for performance evaluation [21–24].

There is one standard benchmarking dataset which primarily contains artificially constructed test cases, i.e., the Software Assurance Reference Dataset (SARD) [25] provided by the National Institute of Standards and Technology (NIST).¹ The dataset has been established for the evaluation of automated defect/vulnerability detection. However, some researchers questioned that the code patterns of these test cases provided by Juliet Test Suite (JTS) [26] from the SARD follow a similar coding format, resulting in a large portion of code appearing identical among many test cases [22]. From this point of view, the code samples in the aforementioned dataset failed to reflect the characteristics of code patterns in the production environment.

It is always beneficial, for both the research and the industrial communities, to have a real-world vulnerability dataset that serves as reliable evaluation metrics for measuring the performance of vulnerability detection

approaches. In this paper, we take a step further in this direction by constructing a function-level vulnerability dataset containing real-world source code samples collected from nine open-source projects. The dataset is constructed by our manually labeling 1471 vulnerable functions based on the disclosed vulnerability records from the Common Vulnerabilities and Exposures (CVE)² and the National Vulnerability Database (NVD),³ which are publicly accessible vulnerability data repositories. Excluding the vulnerable functions and vulnerable files (e.g., some CVE records mention which files are vulnerable but do not specifically describe the exact location of vulnerabilities), we collect a subset of the remaining functions as non-vulnerable functions. The detail of data labeling and collection process will be discussed in Sect. 3.1. For evaluation, we implement a function-level vulnerability detection framework that incorporates six neural network models as a systematic benchmark. It aims to investigate the performance behavior of different network architectures on the constructed dataset and to provide a referenced benchmark for neural model-based vulnerability detection at function-level granularity.

We have made our data and developed framework publicly available at Github⁴ and hope that with the framework as an easy-to-use tool, the dataset can serve as ground truth for ML-based function-level vulnerability detection and a baseline for evaluating the relevant approaches. The contributions of this paper are summarized as follows:

- We construct a real-world vulnerability ground truth dataset at function level. We manually label 1471 vulnerable and collect 59,297 non-vulnerable function samples from nine open-source software projects. Each sample in the dataset is provided in a source code function-label pair.
- We perform a comprehensive evaluation using six mainstream neural networks mentioned in the recent literature. Experiments revealed that in terms of detection performance, the convolutional neural network (CNN) achieves the best performance on our dataset. As to the generalization ability, the Fully Connected Network (FCN) outperforms the other network architectures.
- We develop a vulnerability detection framework with the six mainstream network models built-in, providing one-click execution for model training and testing on the proposed dataset. The framework supports easy-to-implement Application Programming Interfaces (APIs),

² <https://cve.mitre.org/>.

³ <https://nvd.nist.gov/>.

⁴ <https://github.com/Seahymn2019/Function-level-Vulnerability-Dataset>.

¹ <https://www.nist.gov/>.

allowing novel neural network models and newly labeled codes to be easily integrated into the framework.

The rest of this paper is organized as follows: The related studies are reviewed in Sect. 2. Section 3 presents the data labeling and collection criteria and the methodology for comparing and evaluating the network-based vulnerability detection approaches. In Sect. 4, we provide the analysis of the comparative results on our dataset. Section 5 concludes this paper.

2 Related works

Motivated by the success of neural techniques in many areas, such as image and voice recognition, researchers have proposed various approaches which apply neural networks for software vulnerability detection. We categorize existing studies in this field into three categories based on the different network structures used.

The FCN, which is also called Multi-layer Perceptron (MLP), is “input structure-agnostic” [27]. Namely, it can take many forms of input data (e.g., images or sequences), offering researchers the flexibility to use various forms of handcrafted features for learning. A pioneering study proposed by Shar and Tan [28] applied the MLP for detecting SQL injection (SQLI) and cross-site scripting (XSS) vulnerabilities on PHP applications. They extracted features to depict input sanitization patterns from Control Flow Graphs (CFGs) and Data Dependency Graphs (DDGs) and used these features as input to the MLP. Grieco et al. [29] combined static and dynamic analysis to derive features from call traces for detecting memory corruption vulnerabilities using MLP. They assumed that the usage patterns of C library functions could reveal the characteristics of memory corruption vulnerabilities. Dong et al. [30] applied FCN for predicting vulnerabilities in Android binary executable files using features extracted from Dalvik instructions. More recently, Peng et al. [24] used the N-gram model to extract frequency-based features from the surface text of Java source code for training a FCN to identify vulnerable files/classes in Android applications.

However, FCN/MLP is not specifically designed for processing sequentially dependent data such as natural languages, program code or voice data. Network structures catered for learning context-aware information receive more attention in code analysis, particularly in vulnerability detection. The filters of CNN enable the network to learn representations from different sizes of code context, which has been shown to be effective for vulnerability detection tasks. Lee et al. [31] applied CNN for learning vulnerable code patterns from the features extracted from

assembly instructions. Later, Harer et al. [32] and Russell et al. [33] utilized CNN as a feature extractor instead of a classifier. Their experiments showed that using a separate random forest classifier trained by the feature sets learned from the CNN led to better performance compared with using the CNN as a classifier.

The ability to model dependencies in sequences has made Recurrent Neural Networks (RNNs) to be alternative choices for researchers to learn potentially vulnerable code patterns. Based on a program representation called “code gadget,” Li et al. [34] adopted the bidirectional form of the Long Short-Term Memory (LSTM) [35] network (we call it Bi-LSTM) for detecting buffer error and resources management error vulnerabilities. Another line of studies utilized the Abstract Syntax Trees (ASTs) from source code functions to derive features. These features were then fed to the Bi-LSTM network to obtain high-level representations indicative of potentially vulnerable functions [18, 19] or processed by a sequence to sequence (seq2seq) LSTM network to generate refined features for within- and cross-project vulnerability detection [36]. A more systematic approach proposed by Li et al. [37] applied two RNNs (i.e., Bi-LSTM and the bidirectional Gated Recurrent Unit (GRU) [38]) for learning syntactic and semantic characteristics of code from both ASTs and CFGs. Lin et al. [20] also utilized two Bi-LSTM networks for extracting feature representations from two data sources of different types which contain samples from the open-source software project and the synthetic test cases.

In addition to the studies using mainstream network structures for vulnerability discovery, researchers can also customize the network structures to satisfy the requirements of downstream tasks. To detect vulnerable functions at binary-level, Wu et al. [39] used a CNN-LSTM network for classification by adding a convolution layer on top of an LSTM layer. Le et al. [40] proposed a Maximal Divergence Sequential Auto-Encoder (MDSAE) for automated learning of representations from machine instruction sequences. Most recently, a novel memory network structure [41, 42] which equips with extra built-in memory blocks was proposed by Choi et al. [22] for identifying buffer overflow vulnerabilities and was further improved by Sestili et al. [23].

Nevertheless, existing approaches were all based on self-constructed/collected datasets. The absence of the real-world vulnerability datasets is hindering the evaluation of the actual usefulness of the published solutions. Therefore, in this paper, we propose a dataset containing source code functions extracted from open-source projects with labels paired, hoping that this dataset can serve as one of the baseline dataset for evaluation.

3 Vulnerability dataset and comparison methodology

In this section, we describe our proposed dataset from three aspects, including data labeling and collection criteria, naming convention, and the statistics. Then, we present the methodology for comparing neural network-based vulnerability detection approaches.

3.1 Data labeling and collection criteria

Our dataset contains vulnerable and non-vulnerable functions collected from nine open-source software projects written in C programming language. They are Asterisk, FFmpeg, Httpd, LibPNG, LibTIFF, OpenSSL, Pidgin, VLC Player and Xen, as shown in Table 1.

3.1.1 Vulnerable function labeling procedure

The vulnerable functions are labeled based on the information from the NVD and CVE. Figure 1 is an example that illustrates the information of a vulnerability on an NVD web page, showing the description, the name, the version of a software project/package which this vulnerability belongs to, the location, the type, the severity level and consequences if this vulnerability is exploited.

When labeling a vulnerable function, we may face four different situations:

- The vulnerability description provided on an NVD or CVE page specifically mentions which function is vulnerable, as Fig. 1 shows. Accordingly, we download the source code of the corresponding version (e.g., the *LibTIFF 4.0.7*) of the software project from Github and

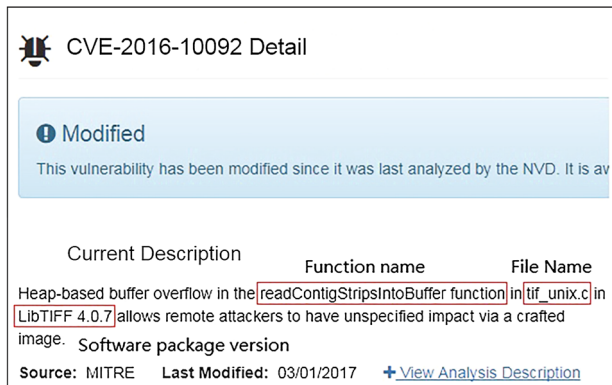


Fig. 1 The screenshot of a vulnerability description on a NVD page, showing the name of a vulnerable function, to which file the function belongs and corresponding version of the software project

locate the vulnerable function (e.g., the *readContigStripsIntoBuffer*) in the file (e.g., the *tif_unix.c*). Then, we label the function and the file as vulnerable. The source code of the vulnerable function is saved as an individual file.

- The vulnerability description does not mention which function is vulnerable. Instead, it mentions the name of the vulnerable file. In this case, we follow the description to mark the file as vulnerable. We also check the Github and search the commit message using the CVE Identifiers (CVE IDs) as the keyword, as a CVE ID uniquely identifies a disclosed vulnerability. Once found, we read through the commit message(s) and identify the one that fixes this CVE. By checking the *diff* files, we acquire the functions related to this CVE fix. The *diff* files enable us to locate the functions prior to the fix and label them as vulnerable. If no commit message that is relevant to the CVE fix is found, we skip this CVE.

Table 1 The name of the nine open-source projects and the number of vulnerable and non-vulnerable functions collected from each of them

Open-source projects	Web page	# of non-vulnerable functions collected	# of vulnerable functions labeled	# of total functions per project
Asterisk	https://www.asterisk.org/	17,755	94	17,849
FFmpeg	https://www.ffmpeg.org/	5552	249	5801
Httpd	https://www.httpd.apache.org	3850	57	3907
LibPNG	http://www.libpng.org/pub/png/libpng.html	577	45	622
LibTIFF	http://www.libtiff.org/	731	123	854
OpenSSL	https://www.openssl.org/	7068	159	7227
Pidgin	https://www.pidgin.im/	8626	29	8655
VLC media player	https://www.videolan.org/vlc/index.html	6115	44	6159
Xen	https://www.xenproject.org/	9023	671	9694
Total		59,297	1471	60,768

- The vulnerability description does not provide any information related to a function or file. In this case, we check the software project's Github and search for CVE-related commit messages as previously mentioned.
- The vulnerability is not associated with any function or file (e.g., the vulnerability caused by misconfigurations and/or incorrect settings). These CVEs are excluded.

In addition to labeling vulnerable functions and files, we also record the CVE IDs, their published/release date, the Common Vulnerability Scoring System (CVSS) Severity, vulnerability type, and file path, etc., for analysis.

3.1.2 Non-vulnerable function collection procedure

Usually, the known defects/vulnerabilities are fixed in the latest release of a software project. Hence, we assume that all disclosed vulnerabilities of the nine open-source projects included in our dataset have also been fixed in their latest version. Based on this assumption, we collect the non-vulnerable functions from the latest version at the time of writing. We exclude the labeled vulnerable functions and the files containing the vulnerable functions identified previously. For the remaining files, we randomly select a subset of them for extracting functions and treat these functions as non-vulnerable. Similar to the way for saving vulnerable functions, we save non-vulnerable functions in C files, keeping each sample in our dataset as an individual file containing only the source code of one function.

3.2 Naming convention

The vulnerable and non-vulnerable functions in the dataset follow the following naming convention—the vulnerable functions are named using the CVE ID and the non-vulnerable functions are named using the name of the function followed by the name of the file to which the function belongs. For example, the *decode_end* function in the *4xm.c* file in the FFmpeg project is named *4xm.c_decode_end.c* in our dataset. This allows us to differentiate functions with identical names. For the cases where one CVE is related to multiple vulnerable functions, we append a numeric as the suffix to the CVE ID. For example, the CVE-2014-0226 is related to two functions (i.e., *status_handler* and *lua_ap_scoreboard_worker*) in the Httpd project. These two functions are named as *CVE-2014-0226_1.c* and *CVE-2014-0226_2.c* in our dataset.

3.3 Dataset statistics

Table 2 lists the annual distribution of the number of vulnerable functions labeled for each software project in our

dataset. Generally, an increasing trend can be seen that over the past 18 years, the number of disclosed vulnerabilities in each project has been growing steadily. Figure 2 illustrates the proportion of different categories (or types) of vulnerabilities in our dataset. The vulnerability categorization is based on the Common Weakness Enumeration (CWE) system. There are 255 buffer errors (CWE-119) vulnerabilities, accounted for the largest proportion. The permissions, privileges and access control (CWE-264) category accounts for the second largest proportion in the dataset and the input validation (CWE-20) is the third. In Fig. 3, the distribution of different severity levels of labeled vulnerabilities is presented. The severity of a vulnerability is measured by the CVSS. The green color refers to the vulnerabilities with a severity score between 0 and 3.6, which is considered as low severity. The yellow color denotes the vulnerabilities of medium severity with scores ranging from 4.0 to 6.9. The red and blue colors represent the high and critical severity with scores ranges from 7.0 to 8.9 and from 9.0 to 10.0, respectively. It can be observed that vulnerabilities with severity scores of 4.3, 5.0 and 6.8 account for the largest proportion. Noticeably, vulnerabilities with severity scores between 7.1 and 10 also occupy a considerable proportion.

3.4 Experimental settings and environment

3.4.1 Neural models

As mentioned before, neural networks are adopted to build the detection models on our dataset. The conventional ML models for vulnerability detection have been studied in the literature [7, 12, 14, 15, 17, 43, 44]. Ghaffarian and Shahriari [11] provided an extensive survey reviewing different approaches that applied non-neural network-based techniques for vulnerability analysis and discovery. Recently, motivated by the breakthrough of deep learning techniques in Natural Language Processing (NLP) and image recognition [45], researchers have adopted these techniques for code analysis, particularly for vulnerability discovery [46].

The application of neural networks for vulnerability detection has achieved promising results [18, 22, 23, 47]. The underlying assumptions are that the programming source code is a logical and semantic structure [46] and the occurrence of vulnerable code fragment usually contains multiple lines of code that are interrelated and contextual [18, 19]. The neural networks, which are hierarchically structured and highly nonlinear, are capable of fitting the patterns that are latent and complex. In particular, the network architectures such as RNNs and CNNs are suitable for processing sequentially dependent and contextual data. On top of these facts, researchers in software

Table 2 The annual distribution of disclosed vulnerable functions in each open-source software project in our dataset (from 2001 to 2018)

Open-source projects	The number of vulnerable functions labeled based on the records from the past 18 years												Total per project
	2001–2007	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	
Asterisk	6	6	4	1	11	10	5	6	5	3	21	16	94
FFmpeg	1	2	1	1	5	50	71	7	23	20	35	33	249
Httpd	14	5	2	4	4	3	2	11	4	1	5	2	57
LibPNG	15	–	3	1	6	5	–	5	5	2	1	2	45
LibTIFF	10	1	3	5	3	4	4	1	0	30	40	22	123
OpenSSL	6	2	–	5	–	3	2	40	32	48	9	12	159
Pidgin	–	1	4	2	4	3	2	6	–	–	7	–	29
VCL media player	3	10	1	2	7	1	2	7	4	2	3	2	44
Xen	–	–	–	–	8	57	95	106	69	86	164	86	671
Total	55	27	18	21	48	136	183	189	142	192	285	175	1471

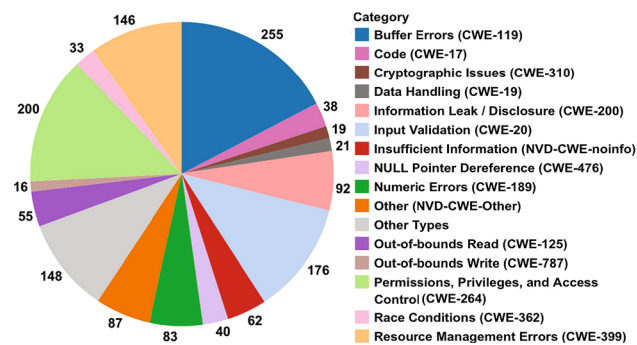


Fig. 2 The pie chart depicts the distribution of vulnerability categories in our dataset. The vulnerability categorization is based on the CWE system. The sectors in the pie chart represent different categories of vulnerabilities and the size of a sector represents the proportion of a specific category of vulnerabilities. The associated numeric values are the numbers of vulnerabilities in the categories

engineering and ML communities have applied various types of neural networks for vulnerability detection.

In this paper, we adopt six types of frequently used network architectures according to the literature for a comprehensive performance evaluation on our dataset. The six network architectures belong to three main categories: (1) FCN; (2) CNN and (3) RNNs. Each category is discussed as follows:

The FCN is a generic structure which is not specifically designed for a particular form of data. It is a typical feed-forward network which consists of at least three fully connected layers (a.k.a dense layers). Usually, the number of neurons in the input layer equals the dimensionality of the input feature vector. The second layer, known as the hidden layer, receives the output from the previous layer as input and produces the output for the next layer. For a neuron in the hidden layer, it takes a weighted sum of its input values from the neurons in the preceding layer. A nonlinear activation function within the neuron is then

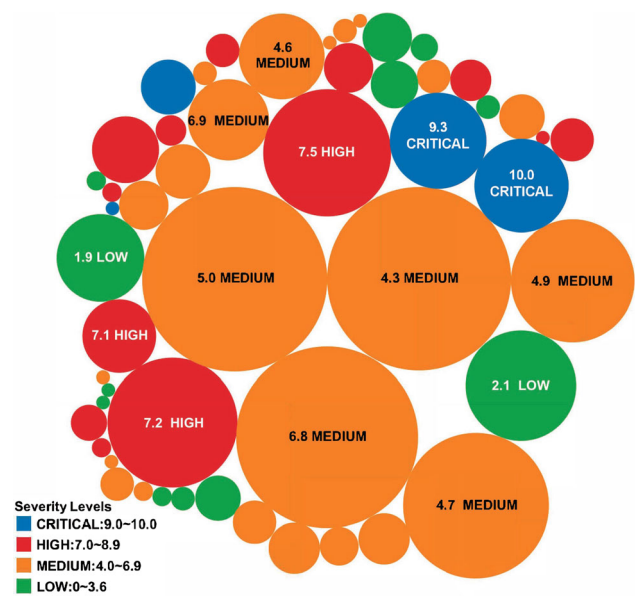


Fig. 3 The bubble chart depicts the distribution of severity levels of vulnerabilities in our dataset. Four colors in the bubble chart correspond with four severity levels of “Low,” “Medium,” “High” and “Critical.” Each level has its own severity score range. The size of a circle represents the proportion of vulnerabilities with a certain severity score

applied to the weighted sum to produce an output which will be propagated to neurons of the subsequent layer.

The FCN we use consists of seven layers as shown in the first row of Table 3. The first layer is a Word2vec embedding layer to convert input sequences to meaningful embeddings (to be discussed in the next sub-section.), followed by a flatten layer. The subsequent five layers are fully connected dense layers to transfer the flatten embeddings to a separable space and merge the sequences to a single probability, indicating whether the corresponding sequence is vulnerable or not.

The RNN different from the FCN, neurons (a.k.a nodes or units) in an RNN have edges (a.k.a the recurrent edges) which connect adjacent time steps to form cycles. For a node with the recurrent edge at a given time step t , its output y_t not only depends on the current input x_t , but also on the hidden node value h_{t-1} which is from the previous time step [48]. This feature enables the RNN to retain information in the past for the current prediction [49], which makes the RNN suitable for tasks requiring contextual processing (e.g., the temporal data). However, training RNNs can be challenging due to the problems of vanishing and exploding gradients encountered when the loss is back-propagated across many time steps [48]. Therefore, variants of RNN, which are the LSTM and GRU, are more commonly used.

The unit of an LSTM network implements a gate mechanism and a memory cell to overcome the vanishing and exploding gradients. An LSTM unit has three gates which are the input, forget and output gates. The gate mechanism and the memory cell work corporately during the training phase to selectively memorize/forget the information [49], so that the information obtained in the preceding sequence can be optionally kept for access when processing the succeeding sequences, which allows the network to learn long-range dependencies. The GRU can be seen as a simplified LSTM unit because the GRU does not have the memory cell and only has two gates: an update gate and a reset gate. The absence of a memory cell and having fewer gates in the GRU enable the network to be trained faster compared with an LSTM network because fewer tensor operations are needed in a GRU [50]. Additionally, due to the GRU network having fewer parameters, it can be an alternative solution to the LSTM network when having a smaller data set [51].

To enhance long-range dependency learning, the bi-directional RNN (e.g., the Bi-LSTM) structure is introduced. A Bi-RNN consists of two different RNNs, capable of processing sequences of both forward and backward directions. This enables the hidden states from each RNN of two opposite directions to be combined (e.g., concatenated) to form a single hidden state, allowing the contextual dependency to be captured. In this paper, we use the LSTM and GRU and their bidirectional forms, i.e., Bi-LSTM and Bi-GRU. In our scenario, they are capable of capturing dependencies among the input sequences, which can be optimized for learning code semantics that require the understanding of code context [18, 47]. In this sense, the occurrence of the vulnerable code fragment, which is usually related to either previous or subsequent code, or even to both, could be captured by RNNs to learn contextually dependent code patterns. As shown in Table 3, the variants of RNNs, listed from the second to the fifth rows, contain one Word2vec embedding layer followed by two RNN layers (either LSTM or GRU) or two bidirectional RNN layers (either Bi-LSTM or Bi-GRU). We use a global max-pooling layer to convert the outputs of the preceding RNN layers. The last two layers are fully connected layers to merge the outputs to a probability value.

The CNN is a feed-forward network that consists of one or more convolution layers, pooling layers and fully connected layers. A CNN is designed to learn the structured spatial data and was originally invented for image processing and computer vision. The convolutional layer of a CNN can be applied to an n -word window (a matrix consists of n -word vectors) to generate features for this context window. Applying this operation to each possible window of n words in a sentence allows the CNN to capture the contextual meanings of the words in a sentence. Followed

Table 3 The settings and configurations of the chosen network architectures, including the number of layers of each network, the applied embedding method, the number of neurons in the first layer, the pooling layer used, and the activation functions applied

Network architecture	Network settings				
	# of layers	Embedding	Layer type (# of neurons in the first layer)	Pooling	Activation function used
FCN	7	Word2vec Embedding	Flatten, dense (128)	No	ReLU, sigmoid
GRU	6		GRU (128), dense (32)	Global max pooling	tanh, ReLU, sigmoid
LSTM			LSTM (128), dense (32)		
Bi-GRU			Forward GRU (64)/Backward GRU (64)		
Bi-LSTM			Forward LSTM (64)/Backward LSTM (64)		
Text-CNN	8		Convolution layers (16 filters) with filter sizes being 3, 4, 5 and 6, dense(64)	2-D Max pooling	ReLU, sigmoid

by convolutions, pooling layers are used to perform downsampling of the feature representations.

CNNs have achieved state-of-the-art performance on many NLP tasks [52–54] and is also applied for learning contextualized code semantics in code analysis tasks [32, 33, 55]. In this paper, we adopt the text-CNN architecture proposed by Kim [52] for building the vulnerability detection model. The difference from the text-CNN implementation is that we downsize the number of filters in the convolution layers so the network only has 16 filters and the filter sizes are 3, 4, 5 and 6, respectively. The detailed configuration is shown in the last row of Table 3.

Table 3 also lists the choice of activation functions and the number of neurons used in each layer for the different network architectures adopted in the experiments. These settings are referenced from the literature which applied neural networks for vulnerability detection (refer to Sect. 2 for details). We further adjust and optimize the settings based on the performance of our initial experiments. For all networks, the stochastic gradient descent (SGD) optimizer is used with its default parameter settings provided by *Keras* [56]. To prevent overfitting, we apply the Dropout [57] with a value of 0.5 to regularize the networks. During the training phase, a relatively small batch size (with the value of 16) is used to achieve better generalization. Every network completes training after 120 epochs.

To deal with the data imbalance issue on our dataset, we adopt the cost-sensitive learning by applying different weights to the “binary cross-entropy” loss function. The weights of the two classes can be calculated by the following equations:

$$\text{weight} = \frac{n_{\text{samples}}}{(n_{\text{classes}} * \text{samples}_{\text{in_oneclass}})} \quad (1)$$

where the n_{samples} refers to the number of samples in total; the n_{classes} is the number of classes and the $\text{samples}_{\text{in_oneclass}}$ denotes the number of samples in one class. According to this formula, the minority class (i.e., the vulnerable class) is assigned with more weights than the majority class (i.e., the non-vulnerable class) during the process of loss minimization.

3.4.2 Feature learning and embedding scheme

Neural models allow code fragments to be directly used for learning, reducing code analysis efforts. In this paper, we directly use the source code as the input to the neural networks. Each sample is a source code function which has been converted to a code sequence. The neural networks take numeric vectors of unified length as input. First, we build a mapping table to link each code element within the code sequence to integers. These integers act as “tokens” which uniquely identify each code element. Second, we

follow the standard practice to handle the sequences of various lengths by applying padding and truncation. Since 90% of vulnerable functions in our dataset containing less than 1000 code elements when converted to sequences, we use 1000 as the truncation threshold to truncate the function sequences having more than 1000 elements down to 1000. For the function sequences with less than 1000 elements, we use zeros for end-of-sequence padding. The padding and truncation mechanism achieves a good balance among length, excessive-sparsity of the converted sequences, and information loss [18].

To preserve the code semantic meanings of input function sequences, we apply Word2vec [58] with the Continuous Bag-of-Words (CBOW) model to convert each element of the sequence to a dense vector of 100 dimensions. With Word2vec, the elements of sequences can be represented by semantically meaningful vector representations so that the code elements share similar contexts in the codebase are clustered in close proximity in the vector space. This enables the neural networks to capture rich and expressive semantics from the codebase, facilitating the learning of potentially vulnerable code patterns. In this paper, we train the Word2vec model using all the source code of the nine open-source projects.

3.4.3 Performance evaluation

To evaluate the proposed dataset and the detection performance of each neural network, we perform experiments on both the C test cases from the JTS provided by the SARD and the real-world C function samples from our proposed dataset. The evaluation of C test cases from the SARD aims to examine the performance of chosen neural networks on detecting the artificially constructed vulnerable function samples. Also, the performance behaviors of different neural networks on the artificial dataset and our proposed real-world dataset can be compared.

To evaluate the performance on artificial function samples from the JTS, we randomly selected 10,000 vulnerable function samples and 10,000 non-vulnerable ones from the test suite to form a dataset. We then partition the dataset into training, validation and test sets with the ratio of 6:2:2 and ensure that the number of vulnerable and non-vulnerable samples are equal in each set. We notice that the vulnerable and non-vulnerable function samples from the JTS contain the text “bad” and “good” as substrings in the names functions, and in the variable and parameter names. To avoid the neural networks being biased by the presence of these substrings, we remove them.

To investigate the overall performance of each neural network on detecting the real-world function-level vulnerabilities, we mix the vulnerable and non-vulnerable functions from all the nine projects together to form a

dataset. We also partition the dataset into training, validation and test sets with the ratio of 6:2:2. To guarantee that the ratio of the vulnerable and non-vulnerable functions in each project is kept in the training, validation and test sets, the partition process is performed first on the data of individual project and then the partitioned training, validation and test sets of each project are merged. This ensures that there is no single project whose data only appear in training, validation or test set.

The cross-project experiments aim to examine the generalization ability of each neural network for detecting vulnerabilities at function level. The cross-project experiments intend to simulate the scenario where a target software project does not have any labeled data for training while there are some historical labeled vulnerability data from other software projects. We use eight out of nine software projects as the training and validation set and one software project as the test set. For example, in the first experiment, we use project FFmpeg as the test set and the data of the other eight projects will be used as the training and validation sets.

3.4.4 Evaluation metrics

To evaluate the performance of chosen neural models on the artificial samples from the JTS, we use the mainstream performance metrics, i.e., precision, recall and F1-score, because there is no data imbalance issue in the artificial dataset. However, in practice, since vulnerability detection mainly relies on manual inspection, it is not cost-effective for a security expert to check every function of a codebase. A practical way is to examine a small portion of code that is most likely to be vulnerable. To help resolve this problem, in this paper, we first retrieve a list of functions ranked by their vulnerable probabilities then use the ranking information as metrics to measure performance. Similar measurements are commonly applied in scenarios of information retrieval by search engines which measure how many relevant documents are acquired in all the top- k retrieved documents [59].

In this paper, we choose the mainstream neural networks to build vulnerability detection models on our dataset. The performance of a chosen network model is measured by the proportion of functions retrieved against the list of k vulnerable functions. The metrics we apply for performance evaluation are top- k precision ($P@K$) and top- k recall ($R@K$). The $P@K$ refers to the proportion of the top- k retrieved functions which are actually vulnerable. The $R@K$ refers to the proportion of actually vulnerable functions returned in the total number of vulnerable functions. Formally, $P@K$ and $R@K$ can be calculated using the following equations:

$$P@K = \frac{TP@k}{TP@k + FP@k}, R@K = \frac{TP@k}{TP@k + FN@k} \quad (2)$$

where $TP@k$ is the true positive samples which are the actual vulnerable functions found by a detection model when retrieving k most probable vulnerable functions; $FP@k$ refers to the false vulnerable functions detected or the false positive when retrieving k most likely vulnerable functions. $FN@k$ refers to the true vulnerable samples undetected or the false negative when retrieving k functions.

3.4.5 Experimental environment

The Keras (version 2.2.4) [56] with a TensorFlow backend (version 1.14.0) [60] is used to implement the neural network models. The Word2vec embedding is provided by the gensim package (version 3.8.0) [61] using all default settings. The experiments are carried out on a Ubuntu Linux system equipped with 32 GB RAM and an NVIDIA GTX TITAN GPU.

4 Evaluation results and analysis

This section presents the results of experiments and provides an in-depth analysis of the results and findings.

4.1 Performance on artificial dataset

Table 4 presents the detection results of chosen neural network models on the artificially created dataset which consists of function samples collected from the JTS. It can be observed that all network models exhibit similar performance behaviors. Particularly, except for the FCN, all the other models achieve 99% F1-score on both the vulnerable and the non-vulnerable classes. This reveals that the vulnerable patterns from artificially created samples are relatively primitive or less diverse. Thus, they can be easily captured and distinguished by the chosen network models from the non-vulnerable patterns.

4.2 Performance of mixed-project

Table 5 lists the top- k precision and top- k recall results of all the chosen neural network models in the scenario of mixed-project detection. It can be observed that the text-CNN achieves the best top- k precision and recall when k ranged from 10 to 200. When retrieving only 10 most probably vulnerable functions, all models found the nine vulnerable functions (having top-10 precision of 90%) except for the Bi-GRU network. When retrieving 200 functions, the text-CNN achieves 51% of precision. That is,

Table 4 Comparative results of chosen neural network models on artificially constructed function samples randomly selected from the JTS provided by the SARD

Network architecture	Vulnerable functions			Non-vulnerable functions		
	Precision (%)	Recall (%)	F1-score (%)	Precision (%)	Recall (%)	F1-score (%)
FCN	99	90	95	91	99	95
GRU	100	99	99	99	100	99
LSTM	98	100	99	100	98	99
Bi-GRU	98	100	99	100	97	99
Bi-LSTM	98	100	99	100	98	99
Text-CNN	99	99	99	99	99	99

The bold refers to the highest performance achieved for a certain group of experimental settings/configurations. It aims to highlight the best performance and make it stand out

Table 5 Comparative results of chosen neural network models in the mixed-project scenario using the function samples from our proposed dataset

Top k	The chosen neural network models (top- k precision top- k recall)					
	FCN	GRU	LSTM	Bi-GRU	Bi-LSTM	Text-CNN
Top 10	90% 3%	90% 3%	90% 3%	80% 3%	90% 3%	90% 3%
Top 50	62% 11%	72% 12%	66% 11%	67% 11%	66% 11%	78% 13%
Top 100	49% 17%	58% 20%	54% 18%	60% 20%	55% 19%	64% 22%
Top 150	42% 21%	51% 26%	50% 26%	52% 26%	51% 26%	59% 30%
Top 200	37% 25%	45% 30%	44% 30%	47% 32%	45% 31%	51% 35%

The bold refers to the highest performance achieved for a certain group of experimental settings/configurations. It aims to highlight the best performance and make it stand out

the model identifies 102 actually vulnerable functions out of the 200 retrieved functions, which accounted for 35% of the total vulnerable functions in the test set. In contrast to the CNN and RNN architecture, the FCN underperforms when retrieving more than 50 functions. In particular, the FCN only achieves 37% precision when retrieving 200 vulnerable functions. The RNN variants and their bidirectional forms achieve better performance than the FCN, but they performed similarly on our dataset. The GRU and Bi-GRU slightly outperform the LSTM and Bi-LSTM, especially when retrieving less than 100 vulnerable functions. There is no significant performance variance observed between the RNN variants and their bidirectional forms.

The results of the mixed-project detection suggest that the capability of obtaining contextual dependency of code sequences facilitates the learning of the vulnerable code patterns. The FCN, which is a generic structure not specifically designed for learning context-aware sequences, detects the fewest vulnerable functions in our dataset. In contrast, the text-CNN and the RNN variants which are capable of learning contextual dependency of code sequences achieves better detection results. The text-CNN achieving the best performance implies that the high-level representations obtained by small code context windows (i.e., the filter implementation with size 3, 4, 5, and 6) could contribute to more effective learning of vulnerable patterns. The RNN variants (i.e., the GRU, the LSTM, and

their bidirectional forms), which are designed for learning long-range dependency of input sequences, underperform compared with the text-CNN on our dataset. In particular, no significant performance improvement is observed when switching the GRU and the LSTM to their bidirectional counterparts. This reveals a fact that not all the vulnerable code patterns are associated with the long-range code context.

Besides, compared with the performance on the mixed-project scenario where the real-world function samples are used, all neural networks performed very well on the artificially constructed dataset. This reveals that the real-world functions from our proposed dataset can more realistically reflect the performance behaviors of neural models in practice, whereas on the artificial dataset, the performance behaviors of different neural networks are indistinctive.

4.3 Generalization ability of cross-project

In the cross-project scenario, we use one software project as the hold-out to form the test set and the other eight as the training set to train the six neural network models. The cross-project experiments produce nine groups of results for the ninth software project, which are listed in Table 6. In contrast to the performance achieved in the mixed-project scenario, significant performance degradation can be

observed for all the chosen networks. This is expected because the data of a software project which is used for testing are unseen by the network models during the training phase.

In terms of the performance of the six chosen networks, the FCN achieves better performance on the majority of the projects compared with the other networks. It can be observed from Table 6 that the other networks’ performance is more project-dependent. For example, the LSTM network achieves the best performance when it is tested on the Httpd project (when the k ranged from 50 to 200) and the text-CNN network achieves the best performance when tested on the Pidgin project. In terms of the performance across the nine software projects, all the six networks

achieved better performance on projects LibTIFF and LibPNG, but they underperform on projects Asterisk, Pidgin and VLC Player.

The cross-project detection results indicate that the FCN architecture demonstrates better generalization ability compared with the text-CNN and the selected RNN variants. Recall the setting of the cross-project experimental setup, the test project data in the cross-project detection scenario are unseen during the network training. Therefore, the more *project-agnostic* knowledge the network model learns from the training projects, the better performance it achieves on the test project. According to Xu [62], the FCNs with the rectifier linear units (ReLUs) activation functions could learn a smooth objective function, leading

Table 6 Comparative results of chosen neural network models in the cross-project scenario using function samples from our proposed dataset. Each project is tested by neural models trained by the other eight projects

Top k	Neural Models	The top- k precision (%) top- k recall (%) of nine open-source software projects								
		Asterisk	FFmpeg	Httpd	LibPNG	LibTIFF	OpenSSL	Pidgin	VLC Player	Xen
Top 10	FCN	2 0	24 1	26 5	28 6	54 4	42 3	8 3	2 0	38 1
	GRU	0 0	18 1	10 2	44 10	66 5	24 2	4 1	0 0	26 0
	LSTM	0 0	22 1	16 3	48 11	52 4	34 2	4 1	2 0	30 0
	Bi-GRU	0 0	38 2	12 2	32 7	50 4	16 1	6 2	0 0	12 0
	Bi-LSTM	0 0	26 1	10 2	42 9	54 4	40 3	2 1	0 0	0 0
	Text-CNN	2 0	22 1	10 2	58 13	50 4	44 3	16 6	2 0	34 1
Top 50	FCN	8 4	33 7	16 14	37 41	56 23	40 12	8 13	3 4	36 3
	GRU	0 0	21 4	12 11	29 32	38 15	19 6	3 5	1 1	18 1
	LSTM	0 0	21 4	18 16	27 30	40 16	25 8	7 12	4 5	20 1
	Bi-GRU	3 1	27 5	10 9	28 31	40 16	19 6	2 3	0 0	9 1
	Bi-LSTM	0 0	22 4	11 10	28 31	41 17	31 10	10 17	2 3	4 0
	Text-CNN	2 1	22 4	8 7	42 47	39 16	25 8	11 19	4 5	20 2
Top 100	FCN	8 9	31 13	12 20	30 68	44 36	33 21	6 21	3 8	41 6
	GRU	0 0	23 9	10 17	21 48	34 28	16 10	5 17	1 2	16 2
	LSTM	0 0	24 9	14 24	19 43	31 26	22 14	7 26	5 10	15 2
	Bi-GRU	3 3	25 10	9 16	23 52	35 28	17 10	4 14	0 0	12 2
	Bi-LSTM	0 0	22 9	9 15	21 47	31 25	24 15	7 26	4 8	6 1
	Text-CNN	3 3	20 8	7 13	30 67	32 26	22 14	7 26	3 6	20 3
Top 150	FCN	8 12	29 17	9 24	24 79	37 45	29 28	5 27	4 12	38 9
	GRU	0 0	23 14	9 23	19 63	30 36	17 16	4 23	1 4	17 4
	LSTM	1 1	24 14	11 29	16 53	26 32	18 17	6 32	4 15	13 3
	Bi-GRU	2 3	25 15	8 22	17 58	30 37	15 14	3 18	1 2	12 3
	Bi-LSTM	0 0	21 13	8 20	18 60	28 34	20 19	7 39	4 15	7 1
	Text-CNN	3 5	19 12	7 18	24 79	31 37	18 17	7 37	4 12	19 4
Top 200	FCN	7 16	27 22	8 28	19 86	33 53	26 33	5 31	4 16	36 11
	GRU	0 1	22 18	9 31	16 72	26 41	15 19	4 27	2 7	16 5
	LSTM	1 2	23 19	9 32	15 66	25 41	16 20	6 41	4 19	12 4
	Bi-GRU	2 3	25 20	8 28	15 68	26 43	13 16	3 21	1 4	12 3
	Bi-LSTM	1 1	21 17	7 25	15 67	25 40	18 23	6 41	4 20	7 2
	Text-CNN	3 6	19 15	6 21	19 85	27 45	17 21	6 44	3 14	9 6

The bold refers to the highest performance achieved for a certain group of experimental settings/configurations. It aims to highlight the best performance and make it stand out

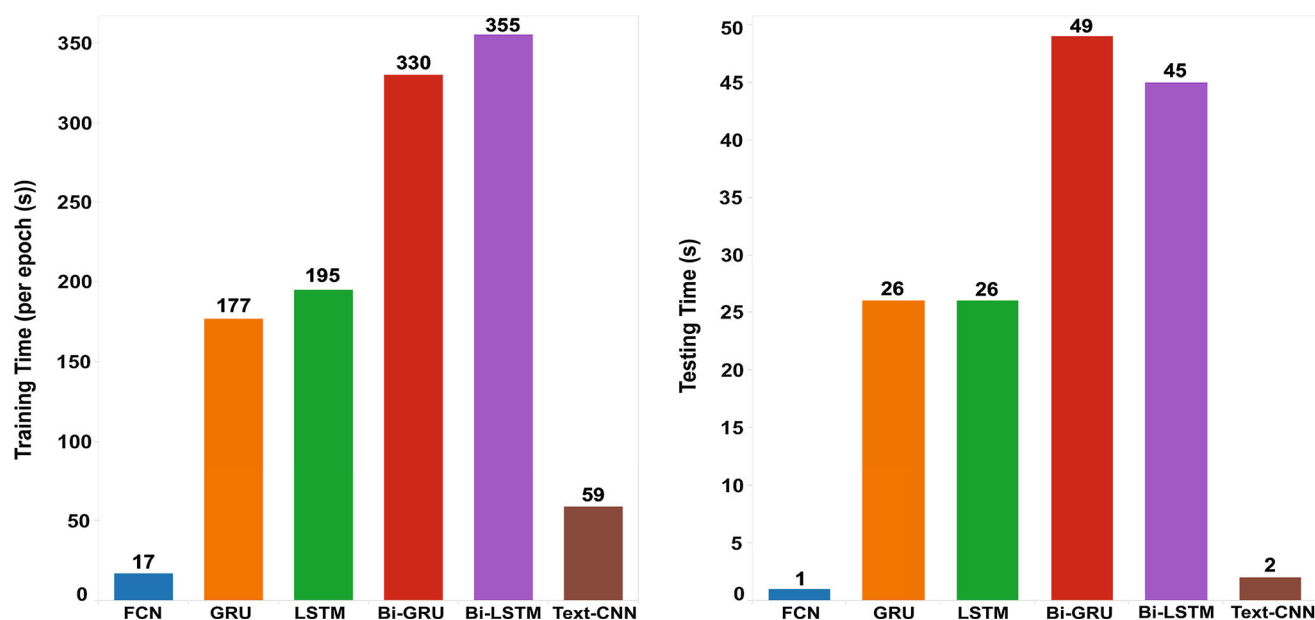


Fig. 4 The comparison of training and test time of the six neural network models, measured in second(s), in the scenario of mixed-project detection. The left bar chart depicts the time of each network

model spent per epoch on average during the training phase. The right one shows the time of each network model used during the test phase

to better generalization ability. For the RNN variants, we speculate that the capability of learning the contextual dependency of the code sequences degrades the generalization ability because the contextual patterns of code in different software projects vary dramatically. Similarly, the generalization ability of the text-CNN is also reduced since the filters applied only learn the local abstraction of the code sequences while the local features are usually more project-specific.

Besides, the fact that detection performance varies on different software projects also suggests that the difference exists among projects. For example, their functionality is one of the factors leading to the variation in detection performance. It is easy to understand that software projects with similar functionalities will exhibit similar code patterns. For example, projects LibPNG and LibTIFF are both image libraries for processing images (e.g., reading, writing image data). When testing on project LibPNG, the network models could learn similar code patterns from project LibTIFF during the training phase, thus resulting in better detection performance. When testing on project LibTIFF, the project LibPNG in the training set allows the network models to capture similar code patterns. Hence, the detection performances on projects LibPNG and LibTIFF are better than that on the other projects. In terms of the degraded performance achieved on projects Asterisk, Pidgin and VLC Player, we attribute this to the high imbalance ratio of vulnerable and non-vulnerable samples in these projects.

4.4 Efficiency

We also compare the efficiency of the six neural network models by measuring the time they spend in the training and testing process. The time is recorded in the scenario of mixed-project detection where the samples in the training, validation and test sets remain unchanged, as 36,458, 12,155 and 12,155 samples, respectively. To speed up the RNN networks, i.e., the LSTM, the GRU and their bidirectional forms, we apply the *CudnnLSTM* and *CudnnGRU* implementations which are customized version of LSTM and GRU based on NVIDIA CUDA Deep Neural Network library.⁵

Figure 4 shows the time variations of the training and testing phases of the six network models. Specifically, the left chart illustrates the average time spent by each network in one epoch during the training process. It can be seen that it takes 17 s for the FCN to complete one epoch. For the text-CNN, it needs approximately 1 min. In contrast, it takes a significantly longer time for the RNN variants even with the Cudnn implementation. In particular, the Bi-LSTM needs around 6 mins to finish one epoch, which is six times slower than the text-CNN. During the testing phase, the time usage pattern of the six models is similar to that of the training phase. The FCN and the text-CNN can complete the test within 1 or 2 s, which is significantly faster than the other networks. The bidirectional LSTM and GRU networks are the slowest, requiring around 50 s to

⁵ <https://developer.nvidia.com/cudnn>.

finish the test. In practice, network models can be trained before use. Generally, among the chosen network models, the text-CNN can be a preferable choice which balances between detection performance and efficiency.

5 Conclusions and future work

In this paper, we have proposed a function-level vulnerability dataset containing real-world source code samples collected from nine open-source projects. We aim to construct this dataset to serve as one of the benchmarking datasets for measuring the performance of function-level vulnerability detection approaches in academia. It can also be used as a code source for neural model-based code analysis tasks such as defect prediction, code generation and code completion. We further evaluate the dataset using three categories of neural networks. Experiments show that the text-CNN achieves the best vulnerability detection performance in the mixed-project scenario and the FCN outperforms the other networks in cross-project detection scenarios. The results indicated that the convolutions which extract the abstract representations of small context windows of codes enable the effective learning of potentially vulnerable code patterns in our dataset and the FCN exhibits the best generalizability in detecting unseen vulnerable functions in software projects that are not involved in training.

We will continue to collect more open-source projects and expand the dataset by labeling more vulnerable functions. We welcome researchers to further improve and contribute to the dataset so it can be beneficial to the state-of-the-art research in the field of vulnerability detection.

Acknowledgements This work was supported in part by the 2019 Educational Research Project of Fujian Province, China, under Grant JAT190695 and in part by the Optoelectronic Information Technology Key Laboratory Open Project Fund of Yunnan Province, China, under Grant YNOE-2020-01.

References

- Equifax had patch 2 months before hack and didn't install it, security group says. <https://www.usatoday.com/story/money/2017/09/14/equifax-identity-theft-hackers-apache-struts/665100001/> September 2017. Accessed 8 June 2019
- Guanjun L, Sheng W, QingLong H, Jun Z, Yang X (2020) Software vulnerability detection using deep neural networks: a survey. *Proc IEEE* 1080(10):1825–1848
- David A (2016) Wheeler. Flawfinder. <https://www.dwheeler.com/flawfinder/> Accessed 20 May 2018
- Cadar C, Dunbar D, Engler DR, et al (2008) Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: *OSDI*, vol 8, pp 209–224
- Sutton M, Greene A, Amini P (2007) Fuzzing: brute force vulnerability discovery. Pearson Education, London
- Newsome J, Song D (2005) Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. Citeseer, Princeton
- Yamaguchi F, Lindner F, Rieck K (2011) Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning. In: *Proceedings of the 5th USENIX conference on Offensive technologies*. USENIX Association
- Nan S, Jun Z, Paul R, Shang G, Zhang Leo Yu, Yang X (2019) Data-driven cybersecurity incident prediction: a survey. *IEEE Commun Surv Tutor* 210(2):1744–1772
- Coulter R, Han Q-L, Pan L, Zhang J, Xiang Y (2019) Data-driven cyber security in perspective-intelligent traffic analysis. *IEEE Trans Cybern.* <https://doi.org/10.1109/TCYB.2019.2940940>
- Jun Z, Yang X, Wang Yu, Wanlei Z, Yong X, Yong G (2013) Network traffic classification using correlation information. *IEEE Trans Parallel Distrib Syst* 240(1):104–117
- Mohammad GS, Reza SH (2017) Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Comput Surv* 500(4):56
- Yonghee S, Andrew M, Laurie W, Osborne Jason A (2011) Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *TSE* 370(6):772–787
- Liu L, De Vel O, Han Q-L, Zhang J, Xiang Y (2018) Detecting and preventing cyber insider threats: a survey. *IEEE Commun Surv Tutor* 200(2):1397–1417
- Yamaguchi F, Golde N, Arp D, Rieck K (2014) Modeling and discovering vulnerabilities with code property graphs. In: *2014 IEEE symposium on security and privacy (SP)*, pp 590–604. IEEE
- Yamaguchi F, Lottmann M, Rieck K (2012) Generalized vulnerability extrapolation using abstract syntax trees. In: *Proceedings of the 28th ACSAC*, pp 359–368. ACM
- Chen X, Li C, Wang D, Wen S, Zhang J, Nepal S, Xiang Y, Ren K (2020) Android HIV: A study of repackaging malware for evading machine-learning detection. *IEEE Trans Inf Forensics Secur* 15:987–1001
- Perl H, Dechand S, Smith M, Arp D, Yamaguchi F, Rieck K, Fahl S, Acar Y (2015) Vccfinder: finding potential vulnerabilities in open-source projects to assist code audits. In: *Proceedings of the 22nd SIGSAC conference on CCS*, pp 426–437. ACM
- Guanjun L, Jun Z, Wei L, Lei P, Yang X, De Vel O, Paul M (2018) Cross-project transfer representation learning for vulnerable function discovery. *IEEE Trans Ind Inf* 140(7):3289–3297
- Lin G, Zhang J, Luo W, Pan L, Xiang Y (2017) Poster: vulnerability discovery with function representation learning from unlabeled projects. In: *Proceedings of the 2017 SIGSAC Conference on CCS*, pp 2539–2541. ACM
- Lin G, Zhang J, Luo W, Pan L, De VO, Montague P, Xiang Y (2019) Software vulnerability discovery via learning multi-domain knowledge bases. *IEEE Trans Depend Secure Comput.* <https://doi.org/10.1109/TDSC.2019.2954088>
- Scandariato R, Walden J, Hovsepian A, Joosen W (2014) Predicting vulnerable software components via text mining. *TSE* 400(10):993–1006
- Choi M, Jeong S, Oh H, Choo J (2017) End-to-end prediction of buffer overruns from raw source code via neural memory networks. *arXiv preprint arXiv:1703.02458*
- Sestili CD, Snaveley WS, VanHoudnos NM (2018) Towards security defect prediction with AI. *arXiv preprint arXiv:1808.09897*
- Peng H, Mou L, Li G, Liu Y, Zhang L, Jin Z (2015) Building program vector representations for deep learning. In: *International conference on knowledge science, engineering and management*, pp 547–553. Springer

25. Black PE (2018) A software assurance reference dataset: Thousands of programs with known bugs. *J Res Natl Inst Stand Technol* 123
26. Black PE, Black PE (2018) Juliet 1.3 Test Suite: Changes From 1.2. US Department of Commerce, National Institute of Standards and Technology
27. Ramsundar B, Zadeh RB (2018) TensorFlow for deep learning: from linear regression to reinforcement learning. O'Reilly Media Inc., Newton
28. Shar LK, Tan HBK (2012) 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
29. Grieco Gustavo, Grinblat Guillermo Luis, Uzal Lucas, Rawat Sanjay, Feist Josselin, Mounier Laurent (2016) Toward large-scale vulnerability discovery using machine learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 85–96. ACM
30. Feng D, Wang LQ, Guoai X, Shaodong Z (2018) Defect prediction in android binary executables using deep neural network. *Wireless Pers Commun* 1020(3):2261–2285
31. Lee YJ, Choi S-H, Kim C, Lim S-H, Park K-W (2017) Learning binary code with deep learning to detect software weakness. In: KSII the 9th international conference on internet (ICONI) 2017 symposium
32. Harer JA, Kim LY, Russell RL, Ozdemir O, Kosta LR, Ranganani A, Hamilton LH, Centeno GI, Key JR, Ellingwood PM et al (2018) Automated software vulnerability detection with machine learning. arXiv preprint [arXiv:1803.04497](https://arxiv.org/abs/1803.04497)
33. Russell R, Kim L, Hamilton L, Lazovich T, Harer J, Ozdemir O, Ellingwood P, McConley M (2018) 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
34. Li Z, Zou D, Xu S, Jin H, Qi H, Hu J (2016) Vulpecker: an automated vulnerability detection system based on code similarity analysis. In: Proceedings of the 32nd ACCSA, pp 201–213. ACM
35. Sepp H, Jürgen S (1997) Long short-term memory. *Neural Comput* 90(8):1735–1780
36. Dam HK, Tran T, Pham T, Ng SW, Grundy J, Ghose A (2017) Automatic feature learning for vulnerability prediction. arXiv preprint [arXiv:1708.02368](https://arxiv.org/abs/1708.02368)
37. Li Z, Zou D, Xu S, Jin H, Zhu Y, Chen Z, Wang S, Wang J (2018) Sysevr: a framework for using deep learning to detect software vulnerabilities. arXiv preprint [arXiv:1807.06756](https://arxiv.org/abs/1807.06756)
38. Kostadinov S (2019) Understanding GRU networks. <https://www.towardsdatascience.com> (December 2017). Accessed 30 Apr 2019
39. Wu F, Wang J, Liu J, Wang W (2017) Vulnerability detection with deep learning. In: 2017 3rd IEEE international conference on computer and communications (ICCC), pp 1298–1302. IEEE
40. Le T, Nguyen T, Le T, Phung D, Montague P, De Olivier V, Qu L (2018) Maximal divergence sequential autoencoder for binary software vulnerability detection
41. Sukhbaatar S, Weston J, Fergus R et al (2015) End-to-end memory networks. In: Advances in neural information processing systems, pp 2440–2448
42. Weston J, Chopra S, Bordes A (2014) Memory networks. arXiv preprint [arXiv:1410.3916](https://arxiv.org/abs/1410.3916)
43. Yonghee S, Laurie W (2013) Can traditional fault prediction models be used for vulnerability prediction? *ESE* 180(1):25–59
44. Wang M, Zhu T, Zhang T, Zhang J, Yu S, Zhou W (2020) Security and privacy in 6G networks: new areas and new challenges. *Digit Commun Netw* 6(3):281–291
45. Vivienne S, Yu-Hsin C, Tien-Ju Y, Emer Joel S (2017) Efficient processing of deep neural networks: a tutorial and survey. *Proc IEEE* 1050(12):2295–2329
46. Miltiadis A, Barr Earl T, Premkumar D, Charles S (2018) A survey of machine learning for big code and naturalness. *ACM Comput Surv* 510(4):81
47. Li Z, Zou D, Xu S, Ou X, Jin H, Wang S, Deng Z, Zhong Y (2018) Vuldeepecker: a deep learning-based system for vulnerability detection. In: Proceedings of NDSS
48. Lipton ZC, Berkowitz J, Elkan C (2015) A critical review of recurrent neural networks for sequence learning. arXiv preprint [arXiv:1506.00019](https://arxiv.org/abs/1506.00019)
49. Olah C (2015) Understanding LSTM networks. GITHUB blog. Accessed 30 Apr 2019
50. Nguyen M (2018) Illustrated guide to LSTM's and GRU's: a step by step explanation. <https://www.towardsdatascience.com>. Accessed 30 Apr 2019
51. Britz D (2015) Recurrent neural network tutorial, part 4 - implementing a GRU/LSTM RNN with python and theano. <https://www.wildml.com>. Accessed 30 Apr 2019
52. Kim Y (2014) Convolutional neural networks for sentence classification. arXiv preprint [arXiv:1408.5882](https://arxiv.org/abs/1408.5882)
53. Zhang Y, Wallace B (2015) A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification. arXiv preprint [arXiv:1510.03820](https://arxiv.org/abs/1510.03820)
54. Yih W-T, He X, Meek C (2014) Semantic parsing for single-relation question answering. In: Proceedings of the 52nd annual meeting of the association for computational linguistics, vol 2, pp 643–648
55. Junyang Q, Jun Z, Wei L, Lei P, Surya N, Yang X (2020) A survey of android malware detection with deep neural models. *ACM Comput Surv (CSUR)* 530(6):1–36
56. Chollet F et al (2015) Keras. <https://github.com/fchollet/keras>
57. Gal Y, Ghahramani Z (2016) Dropout as a Bayesian approximation: representing model uncertainty in deep learning. In: International conference on machine learning, pp 1050–1059
58. Mikolov T, Chen K, Corrado G, Dean J (2013) Efficient estimation of word representations in vector space. arXiv preprint [arXiv:1301.3781](https://arxiv.org/abs/1301.3781)
59. Christopher PR, Manning D, Schütze H (2009) Introduction to information retrieval. Cambridge University Press, Cambridge
60. Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M et al (2016) TensorFlow: a system for large-scale machine learning. *OSDI* 16:265–283
61. Radim R, Petr S (2010) Software framework for topic modelling with large corpora. In: Proceedings of the LREC 2010 workshop on new challenges for NLP frameworks, pp 45–50
62. Xu ZJ (2018) Understanding training and generalization in deep learning by Fourier analysis. arXiv preprint [arXiv:1808.04295](https://arxiv.org/abs/1808.04295)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.