This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2954088, IEEE Transactions on Dependable and Secure Computing

Software Vulnerability Discovery via Learning Multi-domain Knowledge Bases

Guanjun Lin, Jun Zhang*, *Senior Member, IEEE,* Wei Luo, Lei Pan, *Member, IEEE,*
Olivier De Vel, Paul Montague, and Yang Xiang *Senior Member, IEEE*

Abstract—Machine learning (ML) has great potential in automated code vulnerability discovery. However, automated discovery application driven by off-the-shelf machine learning tools often performs poorly due to the shortage of high-quality training data. The scarceness of vulnerability data is almost always a problem for any developing software project during its early stages, which is referred to as the cold-start problem. This paper proposes a framework that utilizes transferable knowledge from pre-existing data sources. In order to improve the detection performance, multiple vulnerability-relevant data sources were selected to form a broader base for learning transferable knowledge. The selected vulnerability-relevant data sources are cross-domain, including historical vulnerability data from different software projects and data from the Software Assurance Reference Database (SARD) consisting of synthetic vulnerability examples and proof-of-concept test cases. To extract the information applicable in vulnerability detection from the cross-domain data sets, we designed a deep-learning-based framework with Long-short Term Memory (LSTM) cells. Our framework combines the heterogeneous data sources to learn unified representations of the patterns of the vulnerable source codes. Empirical studies showed that the unified representations generated by the proposed deep learning networks are feasible and effective, and are transferable for real-world vulnerability detection. Our experiments demonstrated that by leveraging two heterogeneous data sources, the performance of our vulnerability detection outperformed the static vulnerability discovery tool *Flawfinder*. The findings of this paper may stimulate further research in ML-based vulnerability detection using heterogeneous data sources.

Index Terms—Vulnerability detection, Representation Learning, Deep learning.

1 Introduction

Many cybersecurity incidents and data breaches are caused by exploitable vulnerabilities in software [21, 41]. Discovering and detecting software vulnerabilities has been an important research direction. Automated techniques such as rules-based analysis [9, 47], symbolic execution [4], and fuzz testing [42] have been proposed to enhance the vulnerability search. However, these techniques are inefficient when used on a large code base in practice [48]. To improve efficiency, machine learning (ML) techniques were applied to automate the detection of software vulnerabilities and to accelerate the code inspection process.

ML algorithms are capable of learning latent patterns indicative of vulnerable/defective code, potentially outperforming the rules derived from experience, with a significantly improved level of generalization. Nevertheless, the application of traditional ML techniques to vulnerability detection still requires human experts to define features, which largely relies on human experience, level of expertise and depth of domain knowledge [18]. With deep learning, code fragments can be directly used for learning without the need for manual feature extraction and thus relieving experts from the time-consuming and possibly error-prone feature engineering tasks. Recent studies have utilized neural networks for automated learning of semantic feature [45] and high-level representations [18, 20] that could indicate potential vulnerabilities in Java and C/C++ source code.

However, the existing ML-based vulnerability/defect detection approaches, such as [18, 45, 53], have been constructed based on the assumption of the availability of sufficient labeled training data from the homogeneous sources. Unfortunately, this assumption is not always valid. There are no known publicly available software vulnerability repositories that provide real-world vulnerability data with code and label pairs [18, 20]. The relative scarcity of real-world vulnerability data exacerbates the shortcomings of the existing ML-based solutions on real-world software projects, especially for the software projects with a few historical instances of the detected vulnerabilities. Due to the expensive process of manual software vulnerability collection, the lack of training data causes the supervised ML-based vulnerability detection approaches to be challenging to apply. Hence, in practice, manual efforts are still required for the vulnerability discovery task [44].

To enhance the automation of software vulnerability discovery, we propose a deep learning based framework with the capability of leveraging multiple heterogeneous vulnerability-relevant data sources for effectively and automatically learning latent vulnerable programming patterns. On the one hand, the automated learning of vulnerable programming patterns can relieve human experts of the tedious and error-prone feature engineering tasks. On the other hand, the learned latent representations of the vul-

Jun Zhang is the corresponding author.
Guanjun Lin, Jun Zhang, and Yang Xiang are with School of Software and Electrical Engineering, Swinburne University of Technology, Melbourne, VIC 3122, Australia (e-mail: {glin, junzhang, yxiang}@swin.edu.au)
Wei Luo and Lei Pan are with School of Information Technology, Deakin University, Geelong, VIC 3216, Australia (e-mail: {wei.luo, l.pan}@deakin.edu.au).
Olivier De Vel and Paul Montague are with the Defence Science & Technology Group (DSTG), Department of Defence, Australia (e-mail:{Olivier.DeVel, Paul.Montague}@dst.defence.gov.au)

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2954088, IEEE Transactions on Dependable and Secure Computing

nerable patterns from the combination of heterogeneous vulnerability-relevant data sources can be used as useful features to leverage for the shortage of labeled data. A vulnerability-relevant data source is generally one that is publicly available and that includes quasi real-world vulnerability samples as, for example, the vulnerability samples in the Software Assurance Reference Dataset (SARD) project [30]. The vulnerability samples that we used from the SARD project are mostly synthetic test cases presented as proof-of-concept vulnerabilities and patches for a human programmer to learn. We assume that the deep learning algorithms can derive "basic patterns" from the artificially constructed vulnerabilities. The other vulnerability data source includes a limited number of historical vulnerability data instances collected from some popular open-source software projects. By combining the two cross-domain data sources, we design the algorithms to collectively extract the useful information not only from the real-world vulnerability data but also from the synthetic data sets for improving the vulnerability detection performance.

To utilize the heterogeneous data sources, the proposed framework consists of two independent deep learning networks. Each network is trained independently using one of the data sources. Based on the findings of our previous work [20], a Long Short-Term Memory (LSTM) [13] network trained by the historical vulnerability data source could be used as a feature extractor to generate useful features which contain the vulnerable information learned from the vulnerability data source. By using the generated features for training, a classifier can maintain its performance with the lack of labeled data. In this paper, we explore the transfer representation learning capability of a neural network further by learning vulnerable patterns from two vulnerability-relevant data sources. We hypothesize that the source for learning latent vulnerable code patterns should not be limited to the historical vulnerability data source containing real-world software projects. A vulnerability-related data source (i.e., the SARD project) containing the artificial vulnerability samples should also be used as a vulnerability knowledge base. By using two independent neural networks to learn the vulnerability-relevant knowledge from two data sources, respectively, we can combine the learned knowledge to complement the shortage of labeled data and to enhance the vulnerability detection capability.

Firstly, we train two networks using the aforementioned two vulnerability-relevant data sources. Then, both trained networks are used as feature extractors. Given a project with limited labeled data, we feed the data to each trained network for deriving a subset of vulnerability knowledge representations as features. Secondly, we combine the learned representations from each network as features by concatenating the representations. Then, we train a random forest classifier based on the combined representations. Lastly, the trained classifier can be used for detecting vulnerabilities (see Fig. 2). Even for a given project without any labeled data, we can still use one of the trained networks as the classifier for vulnerability detection (see Fig. 1). To ensure the reproducibility, we have publicized our code and the sorted data at Github[1]. In summary, the contributions of this paper are three-fold:

- We propose a deep learning framework utilizing heterogeneous vulnerability-relevant data sources based on two independent deep representation learning networks capable of extracting the useful features for software vulnerable code detection.
- We validate the design of our framework through experiments and demonstrate that using neural networks as feature extractors and a separated classifier to train on the extracted features improve the vulnerability detection performance — a maximum improvement of 60% in precision and 24% in recall was observed.
- Our empirical studies found that the aggregated representations learned by the two independent networks lead to optimal detection performance when compared with the settings of using any single network — a maximum improvement of 5% in precision and 4% in recall was observed. The performance of the proposed framework outperformed *Flawfinder* [47] and our previous work [20]. It implies that the proposed framework can be further extended to cater to multiple data sources.

The rest of this paper is organized as follows: Section 2 provides an overview of the proposed approach and how data sources are collected. Section 3 presents the implementations of the learning of high-level representations from the cross-domain data sources. Our experiments and resulting evaluations are presented in Section 4, followed by a discussion of the limitations of our approach in Section 5. Section 6 lists some related studies, and Section 7 concludes this paper.

2 Approach Overview and Data Collection

This section provides an overview of our proposed framework for software vulnerability detection by describing a workflow of how the code feature representations are learned for vulnerability detection. Following this, we introduce the code data sources and the data collection process.

2.1 Problem Formulation

The proposed method takes a list of functions from a program as input and outputs a function ranking list based on the likelihood of the input functions being vulnerable. Let $\mathcal{F} = \{a_1, a_2, a_3, ...a_n\}$ be all C source code functions (both existing and to-be-developed) in the given software project. We aim to find a function-level vulnerability detector $D : \mathcal{F} \mapsto [0, 1]$, where "1" stands for definitely vulnerable, and "0" stands for definitely non-vulnerable, such that $D(a_i)$ measures the probability of function a_i containing vulnerable code. Often it suffices to treat $D(a_i)$ as a vulnerability score so that we can investigate a small number of functions of the top risk.

1. https://github.com/DanielLin1986/RepresentationsLearningFromMulti_domain

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2954088, IEEE Transactions on Dependable and Secure Computing

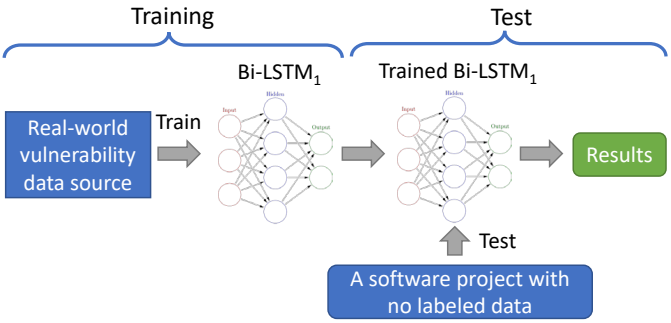


Fig. 1: Scenario 1: In this scenario, a target software project has no labeled data. We train a Bi-LSTM network using the real-world historical vulnerability data from other software projects (the source projects) and feed the target project's code directly to the trained network for classification.

2.2 Workflow

The proposed framework handles different scenarios of the vulnerability detection process. In the first scenario (Scenario 1), we hypothesize that a target software project has no labeled vulnerability data (see Fig. 1). By using transfer learning, our network utilizes the relevant knowledge learned from one task to be applied to a different but related task. In this scenario, we use the historical vulnerability data which are real-world vulnerabilities for training a neural network. We hypothesize that the vulnerable functions of the source projects contain the project-independent vulnerable patterns shared among the vulnerabilities across different software projects and these patterns are discoverable by using the trained neural network for vulnerability detection on a target project.

In the second scenario (Scenario 2), we hypothesize that the target software project has some labeled data available, but the amount of labeled data is insufficient to train a statistically robust classifier. Hence, we exploit the representation learning capability of deep learning algorithm to learn from other vulnerability-relevant data sources, which remedies the shortage of the labeled vulnerability data of the target project.

We divide Scenario 2 into three stages, as depicted in Fig. 2. In the first stage, we train two independent deep learning networks, one for each data source. For the details of the data sources, please refer to Section 2.3. We hypothesize that the trained networks with initialized parameters or weights can capture a broader vulnerability "knowledge" from both vulnerability-relevant data sources. That is, the trained deep networks have learned the hidden patterns in the respective data sources, and the learned patterns from these networks should contain the more generic vulnerability-relevant information than the patterns obtained from an isolated network trained with a single data source. In addition, we found that an alternative solution which uses one single network to learn from both the vulnerability-relevant data sources resulted in suboptimal performance.

The second stage of the scenario obtains the learned patterns or representations from the trained networks. This stage, namely the feature representation learning stage, uses the available labeled data from the *target* software project and feeds them to the two trained deep networks to obtain two groups of representations, respectively. Then, we combine the representations by concatenating them to form an aggregated feature set. For example, we feed a sample to a network trained by one of the vulnerability-relevant data sources. The generated representation by the network is a vector $v_1 = [r_1, r_2, r_3]$. We then feed the same sample to the other network trained by another vulnerability-relevant data source and obtain the representation denoted as the vector $v_2 = [r_4, r_5, r_6]$. The combined feature vector is derived by concatenating v_1 and v_2, that is, $v_{concat} = [r_1, r_2, r_3, r_4, r_5, r_6]$.

In the final stage, we use the remaining data from the target code project as the test set and feed them to each trained network. The obtained representations of the labeled data are used as inputs to train a random forest classifier, and the representations of the test set are fed to the trained classifier to obtain the performance results.

To address the data imbalance between vulnerable and non-vulnerable code samples in the real-world projects, we apply cost-sensitive learning by incorporating different weights of vulnerable and non-vulnerable classes into the objective functions (a.k.a. loss functions) of classifiers used in our experiments. In this paper, we use the equation: $class\_weight = total\_samples/(n\_classes * one\_class\_samples)$ to calculate the weights for each class, where $n\_classes$ is the number of classes, and $one\_class\_samples$ is the number of samples in one class. This equation is based on a heuristic proposed by King and Zeng [17]. That is, the vulnerable class will have a larger weight to penalize the misclassification cost of the vulnerable class more than that of the non-vulnerable class. This setup enables classifiers to overcome the data imbalance challenge during the training phase.

2.3 Data Collection

To overcome the shortage of labeled real-world vulnerability data, we introduce the synthetic vulnerability samples from the SARD project together with real-world vulnerability data sets in our experiments.

2.3.1 The synthetic vulnerability sources from the SARD project

The synthetic vulnerability samples collected from the SARD project were mainly artificially constructed test cases either to simulate known vulnerable source code settings and/or to provide proof-of-concept code demonstrations. In this paper, we only used the C/C++ test cases for our experiments. We developed a crawler to download all of the relevant files. Each downloaded sample is a source code file containing at least one function. According to the SARD naming convention for each test case, the vulnerable functions are named with the phrases such as "bad" or "badSink", and the non-vulnerable ones are with the names containing words like "good" or "goodSink". Therefore, we extracted the functions from the source code files and labeled them as either vulnerable or non-vulnerable according to the SARD naming convention. We hypothesize that the synthetic vulnerable samples contain the proof-of-concept code describing the "basic vulnerability patterns", and that

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2954088, IEEE Transactions on Dependable and Secure Computing

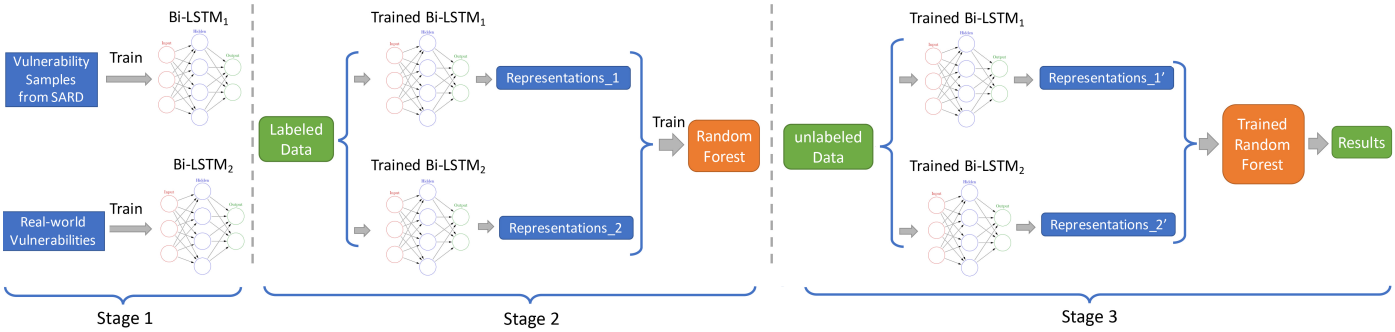


Fig. 2: Scenario 2: In this scenario there are some labeled data available for the target software project. This scenario consists of three stages: the first stage trains two deep learning networks using two data sources; in the second stage, we feed each trained network with the labeled data to obtain two groups of feature representations, and then combine both groups of feature representations to train a random forest classifier; In the third stage, we obtain feature representations by feeding each trained network with the test set (i.e., the unlabeled data) from the target project, and finally use the resulting representations as inputs to the trained random forest classifier to obtain the classification result.

these patterns are discoverable by deep learning algorithms, specifically by the bidirectional LSTM.

2.3.2 The Real-world vulnerability data

We chose to use the real-world vulnerability data source collected by Lin et al. [20] because the granularity of this data source is set at the function level. In this paper, we augmented the data source by adding the vulnerabilities disclosed until April 1, 2018. The data source contains the vulnerable and non-vulnerable functions from the six open-source projects, including FFmpeg, LibTIFF, LibPNG, Pidgin, VLC media player, and Asterisk. The vulnerability labels were obtained from the National Vulnerability Database (NVD) [29] and from the Common Vulnerability and Exposures (CVE) [26] websites. These function-level vulnerability data allows us to build classifiers for function-level vulnerability detection, thus providing a more fine-grained detection capability than that can be achieved at the file- or component-level. Before matching the labels with the source code, we downloaded the corresponding versions of each project's source code from GitHub. Subsequently, each vulnerable function in the software project was manually located and labeled according to the information provided by NVD and CVE websites. Lin et al. [20] discarded the vulnerabilities that spanned across multiple functions or multiple files (e.g., inter-procedural vulnerabilities). Excluding the identified vulnerable functions and the discarded vulnerabilities, they treated the remaining functions as the non-vulnerable ones (see Table 1). By using the function extraction tool, they were able to extract approximately 90% of non-vulnerable functions. We hypothesize that the vulnerable functions in real-world software projects written in the same programming language share generic patterns of the vulnerabilities that are project-agnostic and discoverable by a Bi-LSTM network.

3 Unified Representation Learning

Although both data sources contain source code functions, the samples from the two sources vary in types and complexity. This Section describes how deep learning networks are applied to handle the data sources of different types

TABLE 1: The data sources used in the experiments.

Data source	Dataset/collection	# of functions used/collected	
		Vulnerable	Non-vulnerable
Synthetic samples/Test cases from the SARD project	C source code samples	83,710	52,290
Real-world Open source projects	FFmpeg	213	5,701
	LibTIFF	96	731
	LibPNG	43	577
	Pidgin	29	8,050
	VLC media player	42	3,636
	Asterisk	56	14,648

(e.g., synthetic and real-world vulnerability data), and different processing methods (e.g., ASTs and source code) for learning unified high-level representations with respect to the vulnerabilities of interest. To process the heterogeneous data sources, we feed them to different networks and use one of the hidden layers' output as the learned high-level representations.

3.1 Raw Representations

Before feeding the data sources to the respective networks, the data need to be in a format compatible with deep neural networks. At this stage, we name the data as "raw representations". The data samples from the SARD projects and the real-world vulnerability data sets are source code samples written in the C/C++ languages. Many previous studies, such as [49] and [36], have applied text mining and natural language processing (NLP) techniques for source code-level defect and vulnerability detection. The underpinning assumption is that source code is logical, structural and semantically meaningful. The code can be treated as a "special" language understood by machines (compilers) and communicated by developers, resembling a natural language. This assumption has been formalized by Allamanis et al. [2] as the "naturalness hypothesis". In this paper, we agree that there is a strong semblance between the source code files/functions and "paragraph"/"sentences" found in

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2954088, IEEE Transactions on Dependable and Secure Computing

natural languages.[2] Based on this assumption, we will apply the Recurrent Neural Network (RNN) [25]-based LSTM for processing the source code data.

RNNs are designed for extracting complex interactions residing in sequences. Therefore, RNNs have become powerful tools for solving NLP tasks. Since we hypothesize that a file or a function with a code vulnerability will display certain "linguistic" oddities discoverable by RNNs, we use the RNN-based network for processing the vectors (sequences) computed from the source code and ASTs.

Due to the different lengths between the synthetic samples obtained from the SARD project and the real-world samples from open-source projects, we trim them to a unified length to suit the LSTM network. More specifically, the functions longer than the threshold length will be truncated during their conversion to sequences. On average, the sequences converted from the real-world samples are significantly longer than the ones converted from samples of the SARD project, because the samples from the SARD project only contain the minimal statements for demonstrating a vulnerability. However, for a real-world function, vulnerable statements almost always account for a relatively small portion of the total statements. If we pad the short SARD sequences to the average length of real-world sequences, it will result in adding the excessive redundant information to the SARD sequence so that an LSTM network could begin failing to capture the dependencies in the excessively long input sequences [12, 27]. We further reduced the length of the sequences converted from the real-world samples by discarding some parts of the source code. More precisely, we extracted the ASTs from the source code of the real-world functions and converted the extracted ASTs to sequences.

3.1.1 AST-Level Processing

Using the ASTs extracted from the source code as a means of the representation of the code syntax instead of using the source code itself has been applied by many previous studies [19, 20, 50, 50]. The reason is two-fold: Firstly, ASTs preserve more meaningful syntactic information at the function level than other program representations such as the control flow graph [20]. Secondly, converting the source code functions from real-world projects to ASTs can result in shorter sequences, since an AST only preserves the structural and content-related information while the code punctuations as braces and parentheses are discarded. As per [20], we used the "*CodeSensor*" tool [50] for parsing the source code to ASTs and used the depth-first traversal (DFT) to traverse the ASTs. The combination of the two tools helps us transform the source code into code sequences.

The data samples from the real-world projects are granular at the function level. When converting each function to an AST and subsequently to a sequence, we label each sequence as either vulnerable or non-vulnerable with alignment to the labels of the original function. Thus, we reduce the distinction between the data samples of the SARD project and the real-world projects in terms of the sequence length.

2. Strictly speaking, there are limits to this analogy — e.g., source code can be both highly localized and span long 'distances' (e.g., interprocedural). The extended distance is less likely in the case of natural languages.

3.1.2 Source Code-Level Processing

For the SARD project data set, we use the functions extracted from source code files. During the function extraction process, we remove the comments, spaces, the end-of-line characters, and semicolons, so that only the code content remains. The vulnerable and non-vulnerable functions include the text "bad" and "good" or "bad_sink" and "good_sink" as substrings in the names functions, and often in the variable and parameter names. To avoid ML algorithms being misled by the presence of these substrings, we used a source code obfuscation tool called *Snob* [22] to obfuscate the names of user-self-defined functions, variables and parameters with the randomly generated names. However, the structure of the code, the names of C libraries, API functions, and key words remain unchanged (see the code snippet below). After the code obfuscation process, we convert each function to a textual vector where each element is a word from the original source code function. If a function is vulnerable, we label its converted vector as vulnerable, otherwise, non-vulnerable.

```
1  // The code before obfuscation:
2  void CWE122_Heap_Based_Buffer_Overflow__c_
3  CWE193_char_memcpy_09_bad() {
4      char * data;
5      data = NULL;
6      if (GLOBAL_CONST_TRUE) {
7          data = (char *)malloc(10*sizeof(char));
8          if (data == NULL) {exit(-1);}
9      }
10 //  ...
11 }
12
13 // The code after obfuscation:
14 void C000038CA() {
15     char * C00000A00;
16     C00000A00 = NULL;
17     if (GLOBAL_CONST_TRUE) {
18         C00000A00 = (char *)malloc(10*sizeof(char));
19         if (C00000A00 == NULL) {exit(-1);}
20     }
21 //  ...
22 }
```

The Code sample before and after obfuscation

3.1.3 Padding and Truncation

A fixed input length is required by many ML algorithms. For the sequences converted from the SARD project source code and the real-world vulnerability data source, the sequence length varies significantly. Considering that the 90% of sequences from the real-world vulnerability data source and all the SARD project samples have fewer than 1,000 elements, we have to balance the number of sequence elements and the sparsity of sequences. Hence, we choose 1,000 as the sequence length threshold in consideration of the capacity of LSTM [10, 27]. (Please see Tables 2 and 3 for the statistics on code lengths for each data source). For the long sequences, we truncate their length to 1,000; and the short ones are padded with zeros at the end of sequences. We applied the same length threshold and the same padding/truncation mechanism to the sequences converted from both data sources. However, it is challenging to truncate the actual vulnerable code which is not contained in the source code strings that contributed to the first 1,000 elements in the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2954088, IEEE Transactions on Dependable and Secure Computing

sequence. The data alignment issue will be discussed in Section 5.

At this stage, the samples from both the SARD project and real-world open-source projects have been converted to sequences which can be regarded as textual vectors of uniform length. Because RNNs only take numeric vectors as the inputs, the textual vectors need to be converted to numerical vectors. The conversion is undertaken using a word embedding model described as follows.

3.1.4 Embeddings Training

To preserve the meaning held by each word of the textual vectors, we applied a word embedding to represent each word of the textual vector as a dense vector of fixed dimensions. The word embeddings are distributed representations that incorporate the meaning of a word distributed across multiple components of a vector [2]. According to [2], the application of distributed representations has become a common practice in ML and NLP due to its generalizability. When converting the semantically similar "words" to vectors, we hypothesize that the words having similar semantics in the programming language should be in close proximity in the embedding vector space. To validate this, we visualize some embeddings learned from the trained *Word2vec* [24] model and subsequently project the embeddings into a 2-D plane with the t-Distributed Stochastic Neighbor Embedding (*t-SNE*) [43] algorithm. The t-SNE algorithm is a non-linear dimensionality reduction technique commonly used for visualizing high-dimensional data sets. Fig. 3 shows that different types of elements are grouped into separable clusters. For instance, the type key words "`int`", "`double`", and "`float`" are separated from the operators such as "+", "++", and "<". The functions that are frequently associated with security risks are clustered into different groups, e.g., "`memcpy`" and "`strncpy`" may cause buffer overflow errors, and "`fprint`" and "`snprintf`" may have format string issues.

Specifically, we applied Word2vec [24] with the Continuous Bag-of-Words (CBOW) model to convert each element of the textual vectors to word embeddings of 100 dimensions (which is the default setting). The Word2vec model was trained using the code from both the SARD data source and the real-world open source projects. To ensure that the ML model captured the actual semantics of the code instead of training on possible hints from the comments and the descriptions of the code, we used only the code content for training the Word2vec model.

Fig. 3: A 2-D plot of code vector element embeddings learned with Word2vec. The t-SNE algorithm was used to project 100-dimensional vectors learned using Word2vec and then mapped into a 2-D plane. The blue dots represent the C/C++ data types in the code vector; the green rectangles are operators; the red triangles denote the standard C library functions that are frequently associated with security flaws such as format string errors; the purple rhombi are functions which might be subject to buffer errors, and the black pentagons are the various code delimiters. The figure highlights that the majority of elements with similar code semantics/syntax (e.g., operators, data types, and functions with potential vulnerability risks) are grouped into the same cluster, showing that the use of Word2vec produces meaningful C/C++ source code 'word' embeddings.

3.2 The Deep Network for Representation Learning

Up to this stage, the textual vectors from the SARD project and the real-world open-source projects have been converted to word embeddings which we call the raw representations. They are ready to be fed to deep neural networks to extract high-level representations.

3.2.1 The Bi-LSTM network

The LSTM implementation is a variant of an RNN which uses a gate mechanism and a memory cell, capable of capturing long-term dependencies for the input sequences. Compared with the RNN, it is not subject to the vanishing gradient issue, which leads to more effective model training and better performance [31]. According to an up-to-date design of the LSTM [52], as it is illustrated by Fig. 5 [10], an input x_t to an LSTM unit at the current time step t can be expressed by the following transition equations:

$$\begin{aligned} i_t &= \sigma\left(W^i x_t + U^i h_{t-1} + b^i\right), \\ f_t &= \sigma\left(W^f x_t + U^f h_{t-1} + b^f\right), \\ o_t &= \sigma\left(W^o x_t + U^o h_{t-1} + b^o\right), \\ u_t &= tanh\left(W^u x_t + U^u h_{t-1} + b^u\right), \\ c_t &= i_t \odot u_t + f_t \odot c_{t-1}, \\ h_t &= o_t \odot tanh\left(c_t\right), \end{aligned} \tag{1}$$

where i_t, f_t, and o_t denote input, forget, and output gates, respectively. The notion c_t is a memory cell, h_t is a hidden state, and u_t is an input node. The weight matrix W is the input weight between the input and the current hidden layer, and U which is the recurrent weight between the current and previous hidden layer. Specifically, there are four input weights W^u, W^i, W^f, and W^o, corresponding to the unit input, the input gate, forget, and output gates, respectively. There are four recurrent weights (from the output of the previous unit and the three gates to the current unit), which are U^u, U^i, U^f, and U^o, respectively. The symbol σ and $\tanh$ represent the sigmoid function and the Hyperbolic Tangent

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2954088, IEEE Transactions on Dependable and Secure Computing

TABLE 2: The statistics on code lengths for the real-world vulnerability data source.

Rang of sequence length	<500	>=500 and <1,000	>=1,000 and <1,500	>=1,500 and <2,000	>= 2,000 and <2,500	>= 2,500 and <3,000	>=3,000
# of samples (% of total)	36,421 (91.33%)	2,399 (6.02%)	602 (1.51%)	229 (0.57%)	79 (0.20%)	42 (0.11%)	108 (0.27%)
# of vulnerable samples (% of total vulnerable)	286 (59.71%)	144 (30.06%)	23 (4.80%)	6 (1.25%)	8 (1.67%)	7 (1.46%)	5 (1.04%)

TABLE 3: The statistics on code lengths for the synthetic test cases from the SARD project.

Rang of sequence length	<500	>=500 and <1,000	>=1,000 and <1,500	>=1,500 and <2,000	>= 2,000 and <2,500	>= 2,500 and <3,000	>=3,000
# of samples (% of total)	135,337 (99.51%)	439 (0.32%)	53 (0.04%)	7 (0%)	3 (0%)	4 (0%)	157 (0.12%)
# of vulnerable samples (% of total vulnerable)	52,230 (99.89%)	60 (0.11%)	0	0	0	0	0

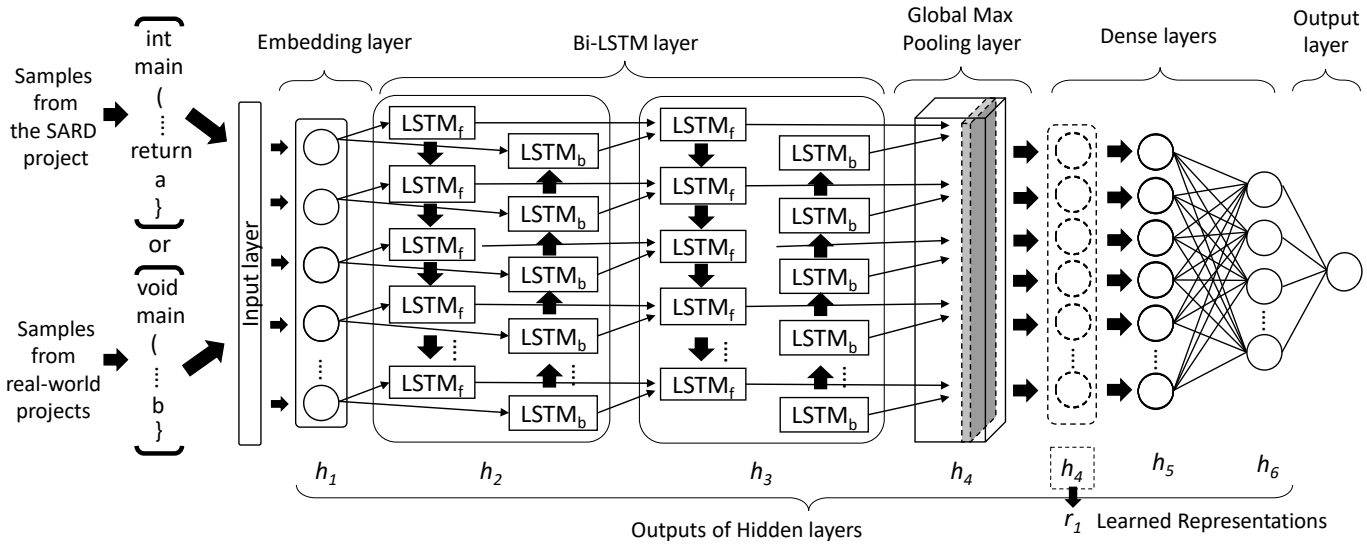


Fig. 4: The structure of the Bi-LSTM deep networks for extracting high-level representations from both the SARD and the real-world vulnerability data sources. The representations can be the output of any hidden layer. In this paper, we used the output of the fourth layer (the global max pooling layer) as the learned feature representations.

function, respectively. The symbol $\odot$ signifies element-wise multiplication. The gate mechanism controls the update of each unit and the state of the memory cell, which allows the network to learn long range dependencies.

Since the occurrence of a vulnerable code fragment usually encompasses a number of code statements, either preceding or subsequent code, or even both, this contextual information is crucial for revealing vulnerable code patterns. To enhance the ability to capture contextual information, we use the LSTM cells as the basic building block and a Bi-LSTM implementation to help detect long-term dependencies in both forward and reverse directions, which can effectively capture the vulnerable programming patterns that are context-dependent. The third hidden layer output h_{3t} of a bidirectional implementation can be calculated using:

$$h_{3t} = z\left(\overrightarrow{h_t}, \overleftarrow{h_t}\right), \tag{2}$$

where $\overrightarrow{h_t}$ and $\overleftarrow{h_t}$ are the values of the hidden layers in the forward and reverse directions, respectively. Function z can be a concatenating, summation, averaging, or multiplication function. In our implementation, we choose to concatenate the outputs of the forward and reverse directions so that the returned tensor will be used for subsequent processing.

The architecture of the Bi-LSTM network is illustrated in Fig. 4. The network takes a mini-batch of vectors (sequences) as an input, and outputs the probabilities of the corresponding input being vulnerable or not. For better generalization, we used a relatively small batch size which was 16 for training. The first layer is an input layer taking a tensor with the shape of $(num\_of\_samples, 1000)$, where the parameter "$num\_of\_samples$" refers to the batch size, and 1,000 is the padding length being the dimension of a sample, also known as the time series in an RNN. The second layer is the Word2vec embedding layer mentioned in Section 3.1.4, which maps each element of input samples to a 100-dimension vector, resulting in a tensor with the shape of $(num\_of\_samples, 1000, 100)$. The third and fourth layers are Bi-LSTM layers, each of which contains 64 LSTM units in a bidirectional form, forming a total of 128 LSTM units. To speed up the Bi-LSTM training process, we used the *CudnnLSTM* implementation which is a customized version of LSTM based on NVIDIA CUDA Deep Neural Network library[3]. The fifth layer is a global max pooling layer which reduces the output dimensionality and aims to retain the most important information. Since each input sample contains at most one vulnerability in our data sets, we surmise

3. https://developer.nvidia.com/cudnn

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2954088, IEEE Transactions on Dependable and Secure Computing

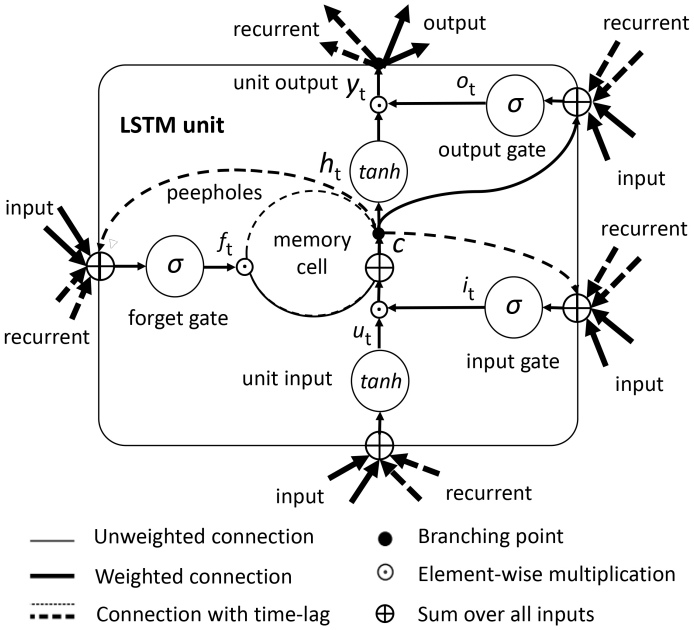


Fig. 5: The structure of an LSTM unit in a hidden layer according to Greff et al. [10]. The notions i_t, f_t, and o_t denote input, forget, and output gates, respectively. The notion c_t is a memory cell, h_t is a hidden state and u_t is an input node. For simplicity, we omit the input and recurrent weights and the bias terms. The notions σ and $\tanh$ denote the Sigmoid and Hyperbolic Tangent function, respectively. The peepholes connections connect the memory cell and the gates are also shown in the figure.

that the vulnerable elements in the input sequence will result in having large values after two layers of the Bi-LSTM. Applying global max pooling for selecting the maximum value can thus help identify potentially vulnerable signals. Let h_i be the i-th hidden layer (see Fig. 4), the global max pooling can be formulated as:

$$h_4 = max_t \left(h_{3t} \right), \tag{3}$$

where the h_{3t} is the output of the third hidden layer, i.e., the output of the bi-LSTM layers, which is a tensor with the shape of $(num\_of\_samples, 1000, 128)$, and the h_4 is a tensor with the shape of $(num\_of\_samples, 128)$. So, the operation of taking a maximum value is on the time series dimension.

The last three layers are dense layers for converging the network to a probability. The first two dense layers contain 128 and 64 neurons, respectively and use the "ReLU" activation function. In our scenario, the detection of a vulnerability is a binary classification. Therefore, the last dense layer applies the "Sigmoid" activation, and the loss function l to be minimized is the "binary cross-entropy":

$$l = -\left(y \log \left(p \right) + \left(1 - y \right) \log \left(1 - p \right) \right), \tag{4}$$

where y is the actual class label for that class (either 0 or 1), and p is the predicted probability of y being either 0 or 1. In this paper, we use the cost-sensitive learning to deal with data imbalance. So, different weights are applied to the two classes when we minimize the loss function l':

$$l' = -\left(w_0 y \log \left(p \right) + w_1 \left(1 - y \right) \log \left(1 - p \right) \right), \tag{5}$$

where w_0 and w_1 are the adjusted weights of two classes, respectively. Our initial experiments had found that using the Stochastic Gradient Descent (SGD) optimizer converges more quickly than other optimizers. To prevent over-fitting, we also implemented dropout (with a value of 0.5) for each layer.

3.2.2 Obtaining Transferable Representations

In the scenario where there are some labeled data, we can process the data and feed them to the networks trained with both the SARD and the real-world vulnerability data sources, and obtain the outputs from one of the hidden layers as the learned representations. We hypothesize that the vulnerable patterns learned from the two data sources can be transferable to generate useful representations for better distinguishing vulnerable from non-vulnerable samples. This process can be formulated as the following equation:

$$r_1 = d \left(x_1 \right), \tag{6}$$

where d is the trained Bi-LSTM network, x_1 is a source code function a_1 (see Section 2.1) that has been converted to a sequence, and r_1 is the hidden layer's output which we call the high-level representations. In this paper, we used the fourth last layer's output (the output of the global max pooling layer) as the representations, so we have $r_1 = h_4$ (see Fig. 4). These representations will be used as features to train a random forest classifier d'. Given a test function sequence ${x_1}'$, we feed ${x_1}'$ to the trained network d and obtain the representation ${r_1}'$. Finally, the representation ${r_1}'$ will be used as the feature set of the test function sequence to the trained random forest classifier d' for classification, as described in the following equations:

$${r_1}' = d \left({x_1}' \right), \; p_1 = d' \left({r_1}' \right), \tag{7}$$

4 Experiments and Outcomes

This section presents a set of experiments conducted on real-world data using our proposed approach for evaluating the effectiveness of the detection of vulnerable code functions.

4.1 Evaluation metrics

The performance of our method is measured by the proportion of vulnerable functions returned in the form of a function list containing the most probable vulnerable functions classified. Hence, the metrics that we apply for performance evaluation are the top-*k* precision (denoted as P@K) and top-*k* recall (denoted as R@K). These metrics are generally used in the context of information retrieval systems such as search engines for measuring how many relevant documents are acquired in all the top-*k* retrieved documents [7], where the relevant documents account for a small portion of the whole document set. In our context, the P@K refers to the proportion in the top-*k* retrieved functions that vulnerable functions are accounted for, and the R@K denotes the proportion of vulnerable functions that are in the top-*k* retrieved functions. For the vulnerable class, the P@K and R@K can be calculated using the following equations:

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2954088, IEEE Transactions on Dependable and Secure Computing

$$P@K = \frac{TP@k}{TP@k + FP@k},\ R@K = \frac{TP@k}{TP@k + FN@k}, \quad (8)$$

where TP@*k* denotes the true positive samples in the *k* returned samples which are most likely vulnerable, meaning the actual vulnerable functions found; FP@*k* and FN@*k* refer to the false positive and false negative samples retrieved in *k* samples, respectively. In practice, the number of vulnerable functions is considerably fewer in number than non-vulnerable ones. When we retrieve the top-*k* functions, we would expect to retrieve as many vulnerable functions as possible. In our experiments, we observe that *k* can vary from 10% to 50% of the total number of retrieved functions. Alternatively, *k* can be an adjustable integer between 10 and 200, to simulate a practical case where the number of functions retrieved is bounded due to the constraints of time and resources.

4.2 Experiment Settings and Environment

4.2.1 Research questions

The experiments are carried out to answer the following Research Questions (RQs):

- RQ1: Are the transfer-learned feature representations generated by the proposed network effective for vulnerability detection?
- RQ2: If the answer of RQ1 is yes, how effective are the feature representations compared with the representations generated without using transfer learning?
- RQ3: Can the network be used as a classifier instead of a feature generator? How would the network perform if it is pre-trained using one of the vulnerability-relevant data sources before used for classification?
- RQ4: Is it necessary to use two networks for learning representations from two heterogeneous vulnerability-relevant data sources? Can a single network achieve comparative results?
- RQ5: How effective is the proposed framework compared with the existing vulnerability detection systems in the two aforementioned scenarios?
- RQ6: How does the proposed framework perform when detecting different types of vulnerabilities?

The first four questions aim to validate our hypotheses and the design of our framework. The last two questions are to evaluate the effectiveness of our framework.

4.2.2 Experiment settings

For comparison with other vulnerability detection systems, we use *Flawfinder* [47] (version 2.0.6) as the baseline system to measure against. *Flawfinder* is a static code analysis tool which takes a source code file as input and outputs the line numbers that contain possible security flaws with warnings, suggestions and flaw types. It is a pattern-based code scanner with well-known vulnerable patterns stored in its database. By comparing the code pattern with the pre-defined vulnerable patterns, the tool raises warnings when the code pattern matches the security flaw pattern. *Flawfinder* is a recognized benchmark used in many previous studies, such as [33] and [18]. Lin et al. [20] also focused on function-level vulnerability detection, where an LSTM network was used for learning vulnerable code patterns. Therefore, we consider their work as another baseline for comparison. We also considered comparing this work with [18]. However, the authors analyzed library/API calls to derive "code gadgets". In general, the "code gadgets" may provide a finer grain detection capability than that at the function-level. Therefore, we decided to exclude the direct comparison between this work and [18].

The detection performance of the proposed framework was evaluated by means of two different groups of experiments to simulate the following scenarios in practice: for a specific software project, there can be 1) no vulnerability labels at all, or 2) only limited vulnerability labels available (in this paper, we assume that the labeled data accounts for 33% of the total). In each scenario, we analyze how the proposed framework performs with respect to RQ4.

When using the real-world open-source vulnerability data source for learning the vulnerable patterns and training on a particular project, we use that project's data as hold-out for test and use the remaining project's data for training to guarantee that the test set (the labeled data) is unseen by the classifier. For instance, when testing on project FFmpeg, we would use projects LibPNG, LibTIFF, Pidgin, VLC, and Asterisk for training. In this paper, our empirical study only uses projects FFmpeg, LibTIFF, and LibPNG as test sets because these software projects contain more vulnerable samples than the other three. Aggregating the information from the enlarged pool of the vulnerable samples contributes to training the classifiers with good performance.

4.2.3 Experiment environment

The Bi-LSTM networks were implemented using *Keras* (version 2.2.4) [5] together with a *TensorFlow* backend (version 1.12.0) [1]. The random forest algorithm was sourced from the *scikit-learn* package (version 0.20.0) [32]. The Word2vec embedding software was provided by the *gensim* package (version 3.4.0) [34] using all default settings. The computational system used was a desktop running Windows-10 with a NVIDIA TITAN Xp GPU.

The random forest is an ensemble classifier that helps to resolve data imbalance issue, therefore was chosen for learning representations extracted by neural networks. The hyperparameter settings of the random forest classifier mainly follow the default settings set by the *scikit-learn* package. But the following customizations are made: The tree's maximum depth is 30; the minimum number of samples required to split an internal node is 4, and the minimum number of samples required to be at a leaf node is 3; the number of trees (i.e., estimators) is set to 8,000. The settings were tuned on our initial experiment and were kept intact throughout the remaining experiments.

4.3 Performance Evaluation

4.3.1 Experiments for answering RQ1

To identify whether the neural network feature representations are effective, we use one baseline approach that applies Word2vec [24] with the Continuous Bag-of-Words

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2954088, IEEE Transactions on Dependable and Secure Computing

TABLE 4: Allocate the vulnerability data to the training and the testing sets. We set 33% of training samples to simulate the case where there is a limited number of labeled vulnerability data.

Open Source Project	Training Set (accounted for 33% of the total)		Test Set	
	# of vulnerable functions	# of non-vulnerable functions	# of non-vulnerable functions	# of vulnerable functions
LibPNG	10	142	346	33
LibTIFF	23	223	492	73
FFmpeg	68	1,638	3,835	145

(CBOW) model for deriving vector representations as features directly from source code and applies a random forest classifier for classification. Since our proposed framework is based on Word2vec embeddings, comparing with this baseline approach helps examine how much the neural network feature representation is contributing to the learning of features indicative of vulnerabilities.

In this group of experiments, we partition the data samples into two parts: 33% training data and 66% test data, as shown in Table 4. When using the Word2vec for retrieving the feature vectors, we feed the training and test sets to the trained Word2vec model which is trained using the code from both the SARD synthetic samples and the real-world open-source projects. The output of the training set will be directly used as features for training a random forest classifier; and the output of the test set will be used as feature fed to the trained random forest classifier to obtain results. For comparison, the same training and test sets will be fed to the two networks for obtaining network representations. One network is trained by the real-world historical vulnerability data source (we name it the CVE source or CVE for short). Another network is trained by the SARD project containing artificially constructed test cases (we name it the SARD for short). In particular, we remove the last three dense layers of the network and obtain the output of the global max pooling layer (the h_4 layer according to Fig. 4) as the learned representations for both the training and test sets.

As shown in Table 5, we report the number of vulnerable functions found in the top-*k*, the top-*k* precision, and the top-*k* recall, where k ranges from 10 to 150 or 200. The random forest classifier trained by the representations obtained from the two networks (trained by the CVE source and the SARD) achieved better detection performance compared to the one trained by the representations obtained from the Word2vec in three software projects. The results showed that the representations generated by the proposed Bi-LSTM network are more effective than the representations directly obtained from the Word2vec model. The networks trained by the two vulnerability-relevant data sources facilitate the learning of representations for vulnerable function detection.

4.3.2 Experiments for answering RQ2

To compare the detection performance of using transfer learning from vulnerability-relevant data sources with the performance when not using transfer learning, we follow the same partition settings according to Table 4. When applying the transfer learning, the training and test set will be fed to neural networks that are trained by both the CVE source and the SARD to obtain feature representations. Then, a random forest classifier will be trained using the feature representations of the training set and test on the test set. In contrast, when not performing transfer learning, we directly use the proposed Bi-LSTM network as the classifier, that is to train the network directly with the training set and perform detection the test set.

As shown in Table 4, the training sets of both LibPNG and LibTIFF projects contain fewer than 300 samples. The Bi-LSTM network did not converge during the training phase, probably due to the training data being small in number. Therefore, we only report the results for the FFmpeg project data set (see Table 6). We report the same performance metrics as we did for the previous experiments. The third row of the Table 6 shows the results obtained from the network trained on the CVE source and the fourth row in the same table lists the results from the network trained on the SARD source. The fifth row reports the results directly using the 33% of the training set to train a Bi-LSTM network, and the trained network was tested on the 66% of the test set without using the transfer learning. The last row shows the results combining the representations (concatenated) from the aforementioned two data sources. One can observe that our method outperformed the method without using transfer learning, especially when combining the representations from two vulnerability-relevant data sources. It proved that the transfer-learned representations generated by the two networks trained on the two vulnerability-relevant data sources were effective feature sets, contributing to better detection performance. In this case, we have assumed that there was one-third of the total samples having labels. Therefore, our method which implements transfer learning can be one of the practical solutions for solving the cold-start issue, as a deep neural network struggles to converge with a small number of labeled data.

4.3.3 Experiments for answering RQ3

Recall that our method uses two neural networks pre-trained on respective vulnerability-relevant data sources for generating representations as features for training separate classifiers. In this group of experiments, we examine the performance of an alternative solution that utilizes neural networks as classifiers. Specifically, we pre-train the two neural networks on respective vulnerability-relevant data sources and fine-tune the networks on a limited set of labeled data. Then, we use both networks as classifiers to test on the test sets of three projects. We set up experiments by partitioning the data set according to Table 4. Subsequently, we train two networks using the CVE source and the SARD. For our method, we feed the trained networks with both the training and test sets before obtaining the fourth last layer's output as the representations and train two random forest classifiers for two groups of representations from two networks. In contrast, we further fine-tune the two networks using the training set before testing the performance.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2954088, IEEE Transactions on Dependable and Secure Computing

TABLE 5: Results of the comparison of using the feature representations directly generated by Word2vec and the representations generated by neural networks (trained by the CVE source and the SARD, respectively) with a random forest classifier on projects FFmpeg, LibTIFF and LibPNG.

Open-source project	Representations generated from	# of vulnerable functions found in the top-*k* functions (top-*k* precision | top-*k* recall)						
		Top 10	Top 20	Top 30	Top 40	Top 50	Top 100	Top 150
FFmpeg	CVE	8 (80%|6%)	10 (50%|7%)	15 (50%|10%)	**23 (58%|16%)**	**28 (56%|19%)**	**47 (47%|32%)**	58 (41%|40%)
	SARD	**9 (90%|6%)**	**14 (70%|10%)**	**19 (63%|13%)**	**23 (58%|16%)**	23 (46%|16%)	43 (43%|30%)	**60 (40%|41%)**
	Word2vec	7 (70%|5%)	12 (60%|8%	16 (53%|11%)	19 (48%|13%)	20 (40%|14%)	30 (30%|28%)	41 (27%|28%)
LibTIFF	CVE	**8 (80%|11%)**	**14 (70%|19%)**	16 (53%|22%)	19 (48%|26%)	21 (42%|29%)	39 (39%|53%)	**50 (33%|68%)**
	SARD	**8 (80%|11%)**	**14 (70%|19%)**	**19 (63%|26%)**	**26 (65%|36%)**	**32 (64%|44%)**	**42 (42%|58%)**	49 (33%|67%)
	Word2vec	5 (50%|7%)	8 (40%|11%)	12 (40%|16%)	14 (35%|19%)	16 (32%|22%)	26 (26%|36%)	39 (26%|53%)
LibPNG	CVE	**10 (100%|30%)**	**18 (90%|55%)**	**26 (87%|79%)**	**27 (68%|82%)**	**30 (60%|91%)**	30 (30%|91%)	32 (21%|97%)
	SARD	9 (90%|27%)	17 (85%|52%)	22 (73%|67%)	24 (60%|73%)	25 (50%|76%)	**33 (33%|100%)**	**33 (22%|100%)**
	Word2vec	9 (90%|27%)	12 (60%|36%)	18 (60%|55%)	22 (55%|67%)	23 (46%|70%)	30 (30%|91%)	31 (21%|94%)

TABLE 6: Results of comparing our method (using transferable representation learning) with using a Bi-LSTM network alone for classification on the FFmpeg project dataset.

FFmpeg Project	# of vulnerable functions found in the top-*k* functions (top-*k* precision | top-*k* recall)							
Representations learned from	Top 10	Top 20	Top 30	Top 40	Top 50	Top 100	Top 150	Top 200
CVE	8 (80%| 6%)	10 (50%| 7%)	15 (50%| 10%)	23 (58%| 16%)	**28 (56%| 19%)**	**47 (47%| 32%)**	58 (41%| 40%)	**70 (35%| 48%)**
SARD	**9 (90%| 6%)**	**14 (70%| 10%)**	19 (63%| 13%)	23 (58%| 16%)	23 (46%| 16%)	43 (43%| 30%)	**60 (40%| 41%)**	65 (33%| 45%)
Classification using Bi-LSTM	4 (40%| 3%)	11 (55%| 8%)	17 (57%| 12%)	18 (45%| 13%)	21 (42%| 15%)	35 (35%| 24%)	47 (31%| 32%)	57 (29%| 39%)
Combined representations	**9 (90%| 6%)**	**14 (70%|10%)**	**20 (67%|14%)**	**24 (60%|17%)**	26 (52%|18%)	46 (46%|32%)	59 (39%|41%)	**70 (35%|48%)**

TABLE 7: Results of the comparison of our method (using neural networks as feature extractors) with using vulnerability-relevant data sources pre-trained neural networks for classification on projects FFmpeg, LibTIFF and LibPNG.

Open-source project	Representations obtained from	# of vulnerable functions found in the top-*k* functions (top-*k* precision | top-*k* recall)						
		Top 10	Top 20	Top 30	Top 40	Top 50	Top 100	Top 150
FFmpeg	CVE	**8 (80%|6%)**	**10 (50%|7%)**	**15 (50%|10%)**	**23 (58%|16%)**	**28 (56%|19%)**	47 (47%|32%)	58 (41%|40%)
	CVE_bilstm	2 (20%|1%)	9 (45%|6%)	13 (43%|9%)	17 (43%|12%)	24 (48%|17%)	**49 (49%|34%)**	**62 (41%|43%)**
	SARD	**9 (90%|6%)**	**14 (70%|10%)**	**19 (63%|13%)**	**23 (58%|16%)**	23 (46%|16%)	43 (43%|30%)	**60 (40%|41%)**
	SARD_bilstm	4 (40%|3%)	9 (45%|6%)	11 (37%|8%)	18 (45%|12%)	**25 (50%|17%)**	**48 (48%|33%)**	**60 (40%|41%)**
LibTIFF	CVE	**8 (80%|11%)**	**14 (70%|19%)**	**16 (53%|22%)**	**19 (48%|26%)**	**21 (42%|29%)**	**39 (39%|53%)**	**50 (33%|68%)**
	CVE_bilstm	4 (40%|5%)	9 (45%|12%)	12 (40%|16%)	16 (40%|22%)	18 (36%|25%)	30 (30%|41%)	42 (28%|58%)
	SARD	**8 (80%|11%)**	**14 (70%|19%)**	**19 (63%|26%)**	**26 (65%|36%)**	**32 (64%|44%)**	**42 (42%|58%)**	**49 (33%|67%)**
	SARD_bilstm	6 (60%|8%)	9 (45%|12%)	15 (50%|21%)	19 (48%|26%)	21 (42%|29%)	32 (32%|44%)	44 (29%|60%)
LibPNG	CVE	**10 (100%|30%)**	**18 (90%|55%)**	**26 (87%|79%)**	**27 (68%|82%)**	**30 (60%|91%)**	**30(30%|91%)**	**32 (21%|97%)**
	CVE_bilstm	7 (70%|21%)	16 (80%|48%)	18 (60%|55%)	20 (50%|61%)	22 (44%|67%)	29 (29%|88%)	31 (21%|94%)
	SARD	**9 (90%|27%)**	**17 (85%|52%)**	**22 (73%|67%)**	**24 (60%|73%)**	**25 (50%|76%)**	**33 (33%|100%)**	**33 (22%|100%)**
	SARD_bilstm	10 (100%|30%)	20 (100%|67%)	24 (80%|73%	26 (65%|79%)	27 (54%|82%)	30 (30%|91%)	32 (21%|97%)

Table 7 lists the comparative results of our method and the method (marked by the names of "CVE_bilstm" and "SARD_bilstm") using pre-trained and fine-tuned neural networks for classification. One can see that our method outperformed the method that uses pre-training and fine-tuning, particularly when retrieving less than 100 functions. One of the reasons is that the separation of the feature learning and classification processes helps resist overfitting. According to the previous studies [11, 35], the classification performance of neural networks was suboptimal compared to the performance achieved by separating the feature learning and classifier training processes due to the data being overfitted on neural networks.

4.3.4 Experiments for answering RQ4

This group of experiments evaluates the necessity of using two networks for learning representations. In this paper, we hypothesize that each network trained using one vulnerable-relevant data source (i.e., the CVE source or the SARD) can capture the vulnerable knowledge of that data source. Two networks trained with two data sources can obtain the knowledge from both sources which can be complementary and can enhance the extraction of vulnerability-indicative representations. However, can a single network trained by both data sources achieve the same goal? To answer this question, we use the source code as the unified input for both the CVE source and the SARD and feed these two sources to one Bi-LSTM network for training. When the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2954088, IEEE Transactions on Dependable and Secure Computing

training completes, we follow the training/test partitions listed in Table 4 and use data samples from projects FFmpeg, LibTIFF and LibPNG for evaluation. Specifically, we feed the training and test sets to the single network trained by both the CVE source and the SARD and obtain the representations. Then, the representations of the training set are used to train a random forest classifier. Lastly, the trained classifier is tested on the representations obtained from the test set for prediction.

Table 8 shows the comparative results between our method which utilizes two networks for learning representations from two data sources and the method using one single network trained on two data sources. The results of our method represented by the name *combined representations* are in the fifth row of each project's result set and the results of the method that uses a single network for obtaining representations are listed in the sixth row, marked by the name *one network*. One can see that our method using two networks trained by two data sources leads to better results in all three projects. One of the reasons is that the two networks trained by the CVE source and the SARD are different despite the network architecture is identical because each trained network learns the knowledge from one source during its training phase. When obtaining representations, we feed the training and test sets to each trained network and the representations extracted contain the knowledge which each network has learned from the respective data source. When we concatenate the representations, the information learned is combined, forming a more abundant knowledge base for the classifier to learn from. In contrast, when using one single network trained on both data sources, the network does learn the knowledge from two data sources, but the knowledge learned is not merely combined. Instead, the network learns the knowledge from a joint distribution of two data sources, which is different from combining the knowledge learned individually from two data source by the two separated networks.

4.3.5 Experiments for answering RQ5 (scenario 1)

In many non-popular open-source projects, it is common to have no labeled vulnerability data, which makes the traditional supervised ML solutions inapplicable. To validate whether the proposed framework is capable of extracting high-level representations as indicators of vulnerabilities, we trained the proposed Bi-LSTM network using the CVE source and tested on three open-source projects: LibPNG, LibTIFF, and FFmpeg, respectively. In this group of experiments, we combined five software projects' data as the training and validation sets and the last project's data as the test set. For instance, when testing on project LibPNG, we used the other five projects' data sets as the historical vulnerability data set for training and validation.

The comparison between our results and the results derived by *Flawfinder* [47] is shown in Table 9. For the LibPNG project, *Flawfinder* returned 93 functions which are regarded as potentially vulnerable according to the risk levels defined by *Flawfinder*. In comparison, our method also retrieved 93 functions from the LibPNG project with a rank list ordered by their probabilities of being vulnerable. Then, we compared the number of vulnerable functions found in the top k retrieved functions, the top k percentage precision, and the top k percentage recall as the metrics for performance comparison. For the projects LibTIFF and FFmpeg, Flawfinder returned 168 and 734 potentially vulnerable functions, respectively. Similarly, we retrieved 168 and 734 potentially vulnerable functions for LibTIFF and FFmpeg, respectively. In terms of detection performance, our framework achieved better accuracy. Table 9 shows that our method outperformed *Flawfinder*, especially for the projects LibPNG and LibTIFF. That is, when measuring the top 10% retrieved functions on LibPNG, our method found 5 vulnerable functions out of a total of 9 retrieved functions (56% precision at top 10% retrieved functions). In contrast, no vulnerable functions were identified by *Flawfinder*. When retrieving 93 functions, our method could identify 82% of total vulnerable functions on LibPNG. In contrast, the *Flawfinder* identified only 30% of the total.

The results suggest that the patterns learned by the proposed Bi-LSTM network from the historical vulnerability samples of software projects (source projects) can be transferred for the detection of vulnerabilities in a target project which has no labeled vulnerability data. With our data sets, the transfer-learned patterns were more useful when compared with the patterns derived from human knowledge. A plausible reason could be that the knowledge-derived patterns are fixed and are too inflexible to cope with the ever-changing code, while the deep neural network is trained on the actual code base so that the patterns learned are more adaptable to variations in the code.

4.3.6 Experiments for answering RQ5 (scenario 2)

In this scenario, we assume that there are a few instances of labeled vulnerability data for a given project. Here we aim to test whether the proposed framework can leverage the limited labeled vulnerability data and achieve a reasonable detection performance. We also use the same open-source projects in this group of experiments, namely: LibPNG, LibTIFF and FFmpeg. We assume that the labeled vulnerability data available for a given project account for one-third of the total data. So, we partition the data according to Table 4. With the same partition settings, we also implemented the method proposed by Lin et al. [20] and use this as the baseline. Although there are some labeled training data available, there is still an insufficient number of vulnerable functions for building a robust statistical model. Therefore, to overcome this limitation, we utilize our proposed framework to process the labeled data for generating effective features in order to optimize the classifier's performance.

Table 8 shows the results when a limited amount of labeled data was used. In general, our method outperformed the baseline method proposed by Lin et al. [20]. The representations obtained from the Bi-LSTM network trained by the SARD source and the CVE source demonstrate their effectiveness for vulnerability detection. It is particularly noticeable for the case of software project LibPNG when combining two groups of representations, retrieving the top 100 functions could identify all of the 33 vulnerable functions in the test set. With project LibTIFF, retrieving the top 150 functions found 50 vulnerable functions which accounted for approximately 68% of total vulnerable functions in the test set. With project FFmpeg, the improvement

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2954088, IEEE Transactions on Dependable and Secure Computing

TABLE 8: Results for the projects LibPNG, LibTIFF and FFmpeg — comparison of our method (representations obtained from two networks) with the method of using one single network and Lin et al. [20].

open-source project	Representations generated from	# of vulnerable functions found in the top-*k* functions (top-*k* precision | top-*k* recall)						
		Top 10	Top 20	Top 30	Top 40	Top 50	Top 100	Top 150
FFmpeg	CVE	8 (80%|6%)	10 (50%|7%)	15 (50%|10%)	23 (58%|16%)	**28 (56%|19%)**	**47 (47%|32%)**	58 (41%|40%)
	SARD	**9 (90%|6%)**	**14 (70%|10%)**	19 (63%|13%)	23 (58%|16%)	23 (46%|16%)	43 (43%|30%)	**60 (40%|41%)**
	Lin et al. [20]	7 (70%|5%)	12 (60%|8%)	16 (53%|11%)	17 (43%|12%)	18 (36%|12%)	30 (30%|21%)	42 (28%|29%)
	One network	**9 (90%|6%)**	11 (55%|8%)	11 (37%|8%)	13 (33%|9%)	16 (32%|11%)	27 (27%|19%)	37 (25%|26%)
	Combined representations	**9 (90%|6%)**	**14 (70%|10%)**	**20 (67%|14%)**	**24 (60%|17%)**	26 (52%|18%)	46 (46%|32%)	59 (39%|41%)
LibTIFF	CVE	**8 (80%|11%)**	14 (70%|19%)	16 (53%|22%)	19 (48%|26%)	21 (42%|29%)	39 (39%|53%)	**50 (33%|68%)**
	SARD	**8 (80%|11%)**	14 (70%|19%)	19 (63%|26%)	**26 (65%|36%)**	**32 (64%|44%)**	**42 (42%|58%)**	49 (33%|67%)
	Lin et al. [20]	9 (90%|12%)	14 (70%|19%)	18 (60%|25%)	19 (48%|26%)	24 (48%|33%)	34 (34%|47%)	39 (26%|53%)
	One network	5 (50%|7%)	10 (50%|14%)	14 (47%|19%)	16 (40%|22%)	17 (34%|23%)	30 (30%|41%)	39 (26%|53%)
	Combined representations	**8 (80%|11%)**	**15 (75%|21%)**	**22 (73%|30%)**	24 (60%|33%)	31 (62%|42%)	41 (41%|56%)	**50 (33%|68%)**
LibPNG	CVE	**10 (100%|30%)**	18 (90%|55%)	**26 (87%|79%)**	**27 (68%|82%)**	**30 (60%|91%)**	30 (30%|91%)	32 (21%|97%)
	SARD	9 (90%|27%)	17 (85%|52%)	22 (73%|67%)	24 (60%|73%)	2 5(50%|76%)	**33 (33%|100%)**	**33 (22%|100%)**
	Lin et al. [20]	6 (60%|18%)	13 (65%|39%)	21 (70%|64%)	24 (60%|72%)	28 (56%|85%)	31 (31%|94%)	32 (21%|97%)
	One network	**10 (100%|30%)**	14 (70%|42%)	20 (67%|61%)	24 (60%|73%)	26 (52%|79%)	30 (30%|91%)	32 (21%|97%)
	Combined representations	**10 (100%|30%)**	**19 (95%|58%)**	**26 (87%|79%)**	**27 (68%|82%)**	27 (54%|82%)	**33 (33%|100%)**	**33 (22%|100%)**

TABLE 9: Results for the open-source software projects LibPNG, LibTIFF and FFmpeg — comparison of our method with *Flawfinder* (based on the risk level larger or equal to 1) if there is no labeled data.

Open-source project	Detection system	# of functions retrieved by *Flawfinder*	# of vulnerable functions found in top-*k* percentage (top-*k* percentage precision | top-*k* percentage recall)		
			Top 10%	Top 50%	Top 100%
LibPNG	*Flawfinder*	93	0	2 (4% | 6%)	10 (11% | 30%)
	This Method		5 (56% | 15%)	19 (55% | 58%)	27 (35% | 82%)
LibTIFF	*Flawfinder*	168	0	3 (4% | 4%)	19 (11% | 26%)
	This Method		5 (30% | 34%)	25 (30% | 34%)	47 (30% | 64%)
FFmpeg	*Flawfinder*	734	1 (1% | 1%)	1 (0% | 1%)	60 (8% | 41%)
	This Method		23 (32% | 16%)	69 (19% | 48%)	109 (15% | 75%)

brought about by combining the two groups of representations was not as significant as the other two projects when returning more than 50 functions. In general, when combining the representations learned from the two networks trained by the two data sources, the performance improved when compared with the use of an individual group of representations. However, in the cases where the combined results were not improved, they were either very close or equal to the upper bound of results achieved by the two networks. The results support our hypothesis that the knowledge contained in the vulnerability-relevant data sources can be learned using our proposed networks, and the knowledge obtained from the individual data sources can be combined to enhance the vulnerability detection.

4.3.7 Experiments for answering RQ6

In this paper, we explored the performance of our proposed framework on detecting the two types of vulnerabilities — Buffer Errors (CWE-119) and Numeric Errors (CWE-189), because these two types of vulnerabilities accounted for the majority of our test sets. Based on the settings used in Scenario 2 (using 33% of training data), we calculated the top *k* recall of each type of vulnerability for the three software projects: LibPNG, LibTIFF and FFmpeg.

A graph depicting the top *k* recall trends of these two types of vulnerabilities (CWE-119 and CWE-189) is shown in Fig. 6. The six lines are grouped into three separate pairs. Each pair corresponds to a particular software project. For the LibPNG project, the two types of vulnerabilities could be quickly found (where retrieving 100 functions, it was able to identify all of the CWE-119 and CWE-189 vulnerabilities in the test set), compared with the other two projects (LibPNG and FFmpeg). This graph suggests that the detection difficulty varies across the projects. In contrast, the results show that the detection difficulty was similar across the two vulnerability types for a particular software project. The graph also indicates that project LibPNG benefits more from the transfer-learned feature representations, compared with the other two projects.

Undeniably, the differences of projects in terms of their functionalities can result in the difference in detection performance. Projects LibTIFF and LibPNG are similar in terms of inputs and outputs. Both projects take input data sequences as hex values, and both output a graphical representation based on the decoded 2-D matrix. Nevertheless, the structure of code and logic can vary significantly. For the project FFmpeg, it shares fewer codes with common "structural features" with the other two projects because FFmpeg handles specific tasks such as video transcoding,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2954088, IEEE Transactions on Dependable and Secure Computing

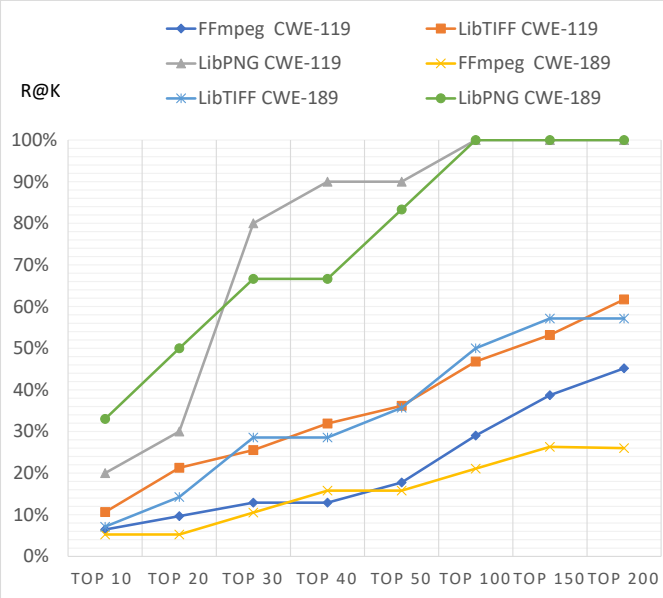


Fig. 6: The top-*k* recall trends of CWE-119 and CWE-189 vulnerabilities on the test sets of the three projects: LibPNG, LibTIFF and FFmpeg in Scenario 2.

video/audio format conversion, and video/audio editing. However, how code similarities/differences affect the detection performance still remains an open research problem.

5 Discussion and Limitations

Our proposed framework has several limitations which can be further investigated. Firstly, in terms of user-friendliness, our deep learning-based method lacks end-user-level interpretability. When presenting detection results, our framework can only provide a ranked list of possible vulnerable functions based on a vulnerability likelihood ranking, without explanation or interpretation. To improve this, we can apply the neural network with attention mechanism for learning how much "attention" the network should be paid to the specific extent of a code sequence which potentially leads to a vulnerability. By analyzing the weights allocated by the attention model to a particular subset of a code sequence, we could understand the relative importance of each attentive subset of code sequences leading to a prediction, which provides interpretability. Enhancing the explainability of the trained neural network models will be our future work.

Secondly, from the perspective of code analysis, our method falls into the static code analysis category, which means that the proposed method cannot detect vulnerabilities that manifest themselves during the code's execution. Also, the method cannot be used when the source code is unavailable. In addition, as our framework aims to identify the vulnerable code snippets within the code function boundary (i.e., at the intra-procedural level), our method cannot detect the vulnerabilities spanning multiple functions or multiple files (i.e., at the inter-procedural level). The inability of tracking variables passing through multiple functions or even files may result in false negatives or false positives. One of the possible solutions is to apply a static taint analysis by monitoring the flow of untrusted variables.

Finally, when converting the ASTs and source code functions to sequences (vectors), the truncation process with an upper limit of 1,000 might discard the actual vulnerable code parts. However, after the embedding process, the vulnerable vectors that have more than 1,000 elements account for approximately one-tenth of the total vulnerable vectors. It is possible that some vulnerable samples whose actual vulnerable contents have been truncated. Eventually, these samples with missing components affect the classification performance; on the other hand, the classification performance degrades after reaching the maximum capacity of LSTM of 1,000 discrete time steps [12, 27]. Practically, the threshold of truncation length should be adjustable according to different data sets. A data set containing mostly long sequences should require a large truncation threshold to trade off the data sparsity and possible information loss. In future work, we will apply a variant LSTM proposed by Neil et al. [27] for processing long sequences converted from the excessively long functions with vulnerable codes in the real world.

6 Related Work

Software vulnerability detection has received much attention from the research community. This section reviews the previous work undertaken in this area.

Early work on code inspection for detecting vulnerabilities relied mainly on rules derived from skills and experience of knowledgeable individuals. Engler et al. [9] were the first to implement rules to automatically identify software bugs and vulnerabilities. Rules that were used included "a pointer that needs to be dereferenced should be non-null", and "calling to a lock function indicates that an unlock function should be called afterwards". Code that did not conform to the rules or best practices could then be identified as potentially vulnerable. Similarly, some static code analysis tools were based on pre-defined rule sets that were derived from well-known security issues, such as *Flawfinder* [47], and *RATS* [40]. However, it is infeasible to design rules to cover all possible programming flaws. The authors of these tools have to keep adding new rules to the tools' database to cope with the ever-changing code base.

Many code clone detection studies for identifying defects/vulnerabilities mainly relied on manually-defined "patterns" for code similarity detection. Instead of comparing with the original code, similarity comparison can be conducted on the converted "patterns" to speed up computation. *Deckard* [15] converted code to tree representations. Then, the tree patterns were further converted to vectors so that the similarity calculation was based on the Euclidean distance of vectors. *Redebug* [14] extracted token strings from source code which were then used to form different sets. Then, the similarity was computed using the two sets, which resulted in improved efficiency. *VUDDY* [16] converted source code functions to strings and compared the hashes of two strings, which achieved significant scalability. Undeniably, code clone detections fail to detect defective/vulnerable code which has been restructured during the cloning process and, more significantly, are not able to identify unknown defects/vulnerabilities.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2954088, IEEE Transactions on Dependable and Secure Computing

Despite the application of ML to vulnerability detection enabling the automated learning of the implicit vulnerable programming patterns, many conventional ML approaches still heavily rely on security experts to manually define metrics or features which the algorithms can learn from. Early studies applied code complexity metrics such as cyclomatic complexity [23] or, the lines of code as features for vulnerability detection/prediction [6, 38, 39, 46]. Later, Neuhaus et al. [28] extracted features from imports and function calls for predicting vulnerable components of Mozilla software. Perl et al. [33] applied code-based metrics and meta-data retrieved from Github projects to predict commits that lead to vulnerabilities. Yamaguchi et al. [49] applied API usage patterns as features for extrapolating potentially vulnerable functions. Shar et al. [37] utilized hybrid code analysis techniques to derive features from input validation and input sanitization code patterns. The authors also considered the situation where the labeled vulnerability data was insufficient. They applied a semi-supervised solution for solving this problem. However, feature sets used by the aforementioned studies are primarily task-specific, therefore, are not generalizable.

To further relieve experts of tedious and possible error-prone feature engineering tasks, deep learning-based solutions have previously been adopted for defect/vulnerability detection. It is hypothesised that the deep learning algorithms are capable of extracting high-level representations that can be more effective than the features driven by human knowledge. Wang et al. [45] leveraged Deep Belief Networks (DBN) to extract semantic features from ASTs obtained from Java source code for file-level defect detection. Lin et al. [20] applied an LSTM to learn semantic representations from historical vulnerable data from open source software projects, focusing on function-level vulnerability detection. Li et al. [18] proposed a program representation called "code gadget" for depicting data or control dependencies. They also used an LSTM network based on code gadgets for identifying vulnerable code fragments, which provided a finer detection granularity.

However, Li et al. [18] focused on detecting two specific types of vulnerabilities, and their performance results were not entirely derived from the real-world data. Our work takes advantages of vulnerability-relevant data sources, which greatly extends the usable data count, thus providing better performance on real-world vulnerability data sets.

In contrast from the aforementioned approaches, the dynamic analysis of code, such as black-box fuzzing [8], does not need any program analysis, thus no rules/patterns are required to be generated from the code base or the underlying system to guide the test. Nevertheless, black-box fuzzing may suffer from low code coverage and can be inefficient on a large code base. The gray-box fuzzing approaches, such as AFL [3, 51], have increased the code coverage and efficiency, while still not requiring sufficient program/code analysis nor knowledge-based rules to facilitate the code analysis.

7 Conclusion

In this paper, we proposed a framework to solve the cold-start problem that many ML-based vulnerability detection approaches generally face in practice. Specifically, the cold-start problem refers to the issue of the shortage or even the lack of ground truth vulnerability data during the early stages of software project development, or as found in the case of many less popular open source projects. To tackle this problem, our framework leverages the representation learning capability of deep learning algorithms for automatically extracting high-level representations that are indicative of vulnerable patterns from vulnerable-relevant data sources, despite these sources containing neither entirely real-world vulnerabilities nor samples closely related to the target project. To bridge the differences among these cross-domain data sources, we designed a Bi-LSTM network for handling data of different types and formats and generated unified representations which are effective for indicating the potentially vulnerable functions on our data set. Our empirical study showed that the knowledge contained in the vulnerable-relevant data sources could be transferred to facilitate the learning of representations for real-world code vulnerability detection. Our results also showed that even without labeled data, our proposed framework can outperform the baseline tool *Flawfinder* using our test software projects.

Acknowledgment

This research is supported by the NVIDIA Corporation with the donation of the NVIDIA TITAN Xp GPU.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[2] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 81, 2018.

[3] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 SIGSAC Conference on CCS*. ACM, 2016, p. 1032–1043.

[4] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[5] F. Chollet *et al.*, "Keras," https://github.com/fchollet/keras, 2015.

[6] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *JSA*, vol. 57, no. 3, pp. 294–313, 2011.

[7] P. R. Christopher D. Manning and H. Schütze, *Introduction to Information Retrieval*. Cambridge University: Cambridge University Press, 2009.

[8] F. Duchene, R. Groz, S. Rawat, and J.-L. Richier, "Xss vulnerability detection using model inference assisted evolutionary fuzzing," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 2012, pp. 815–817.

[9] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *SIGOPS Operating Systems Review*, vol. 35, no. 5. ACM, 2001, pp. 57–72.

[10] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "Lstm: A search space odyssey," *IEEE transactions on neural networks and learning systems*, vol. 28, no. 10, pp. 2222–2232, 2016.

[11] J. A. Harer, L. Y. Kim, R. L. Russell, O. Ozdemir, L. R. Kosta, A. Rangamani, L. H. Hamilton, G. I. Centeno, J. R. Key, P. M. Ellingwood *et al.*, "Automated software vulnerability detection with machine learning," *arXiv preprint arXiv:1803.04497*, 2018.

[12] S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber *et al.*, "Gradient flow in recurrent nets: the difficulty of learning long-term dependencies," 2001.

[13] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[14] J. Jang, A. Agrawal, and D. Brumley, "Redebug: finding unpatched code clones in entire os distributions," in *Security and Privacy (SP), 2012 Symposium on*. IEEE, 2012, pp. 48–62.

[15] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 96–105.

[16] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *Security and Privacy (SP), 2017 Symposium on*. IEEE, 2017, pp. 595–614.

[17] G. King and L. Zeng, "Logistic regression in rare events data," *Political analysis*, vol. 9, no. 2, pp. 137–163, 2001.

[18] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," in *Proceedings of NDSS*, 2018.

[19] G. Lin, J. Zhang, W. Luo, L. Pan, and Y. Xiang, "Poster: Vulnerability discovery with function representation learning from unlabeled projects," in *Proceedings of the 2017 SIGSAC Conference on CCS*. ACM, 2017, pp. 2539–2541.

[20] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, O. De Vel, and P. Montague, "Cross-project transfer representation learning for vulnerable function discovery," *IEEE Transactions on Industrial Informatics*, 2018.

[21] L. Liu, O. De Vel, Q.-L. Han, J. Zhang, and Y. Xiang, "Detecting and preventing cyber insider threats: a survey," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 2, pp. 1397–1417, 2018.

[22] MacroExpressions, "Macroexpressions products: Snob – a simple name obfuscator," http://www.macroexpressions.com/snob.html, accessed: 2019-05-22.

[23] T. J. McCabe, "A complexity measure," *TSE*, no. 4, pp. 308–320, 1976.

[24] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[25] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur, "Recurrent neural network based language model," in *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

[26] "The common vulnerability and exposures (cve)," https://cve.mitre.org/index.html, MITRE, accessed: 2019-05-22.

[27] D. Neil, M. Pfeiffer, and S.-C. Liu, "Phased lstm: Accelerating recurrent network training for long or event-based sequences," in *Advances in neural information processing systems*, 2016, pp. 3882–3890.

[28] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proceedings of the 14th Conference on CCS*. ACM, 2007, pp. 529–540.

[29] NIST, "National vulnerability database (nvd)," https://nvd.nist.gov/, accessed: 2019-05-22.

[30] ——, "Software assurance reference dataset project," https://samate.nist.gov/SRD/, accessed: 2019-05-22.

[31] C. Olah, "Understanding lstm networks," *GITHUB blog, posted on August*, vol. 27, p. 2015, 2015.

[32] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *JMLR*, vol. 12, pp. 2825–2830, 2011.

[33] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, "Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proceedings of the 22nd SIGSAC Conference on CCS*. ACM, 2015, pp. 426–437.

[34] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Valletta, Malta: ELRA, May 2010, pp. 45–50, http://is.muni.cz/publication/884893/en.

[35] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2018, pp. 757–762.

[36] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *TSE*, vol. 40, no. 10, pp. 993–1006, 2014.

[37] L. K. Shar, L. C. Briand, and H. B. K. Tan, "Web application vulnerability prediction using hybrid program analysis and machine learning," *IEEE Transactions on dependable and secure computing*, vol. 12, no. 6, pp. 688–707, 2014.

[38] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *TSE*, vol. 37, no. 6, pp. 772–787, 2011.

[39] Y. Shin and L. Williams, "Can traditional fault prediction models be used for vulnerability prediction?" *ESE*, vol. 18, no. 1, pp. 25–59, 2013.

[40] S. Software, "Rough audit tool for security," https://code.google.com/archive/p/rough-auditing-tool-for-security/, accessed: 2019-05-22.

[41] N. Sun, J. Zhang, P. Rimba, S. Gao, L. Y. Zhang, and Y. Xiang, "Data-driven cybersecurity incident prediction: A survey," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1744–1772, 2019.

[42] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[43] L. van der Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of machine learning research*, vol. 9, no. Nov, pp. 2579–2605, 2008.

[44] D. Votipka, R. Stevens, E. M. Redmiles, J. Hu, and M. L. Mazurek, "Hackers vs. testers: A comparison of software vulnerability discovery processes," in *Security and Privacy (SP), 2018 Symposium on*. IEEE, 2018, pp. 134–151.

[45] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 297–308.

[46] S. Wen, M. S. Haghighi, C. Chen, Y. Xiang, W. Zhou, and W. Jia, "A sword with two edges: Propagation studies on both positive and negative information in online social networks," *IEEE Transactions on Computers*, vol. 64, no. 3, pp. 640–653, 2014.

[47] D. A. Wheeler, "Flawfinder," https://www.dwheeler.com/flawfinder/, accessed: 2019-05-22.

[48] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Security and Privacy (SP), 2014 Symposium on*. IEEE, 2014, pp. 590–604.

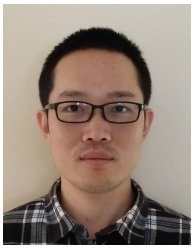
[49] F. Yamaguchi, F. Lindner, and K. Rieck, "Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning," in *Proceedings of the 5th USENIX conference on Offensive technologies*. USENIX Association, 2011.

[50] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proceedings of the 28th ACSAC*. ACM, 2012, pp. 359–368.

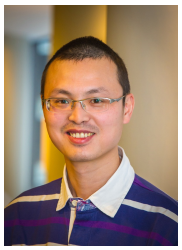
[51] M. Zalewski, "American fuzzy lop (afl)," http://lcamtuf.coredump.cx/afl/, accessed: 2019-05-22.

[52] W. Zaremba and I. Sutskever, "Learning to execute," *arXiv preprint arXiv:1410.4615*, 2014.

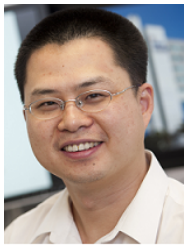
[53] J. Zhang, Y. Xiang, Y. Wang, W. Zhou, Y. Xiang, and Y. Guan, "Network traffic classification using correlation information," *IEEE Transactions on Parallel and Distributed systems*, vol. 24, no. 1, pp. 104–117, 2013.



Guanjun Lin received the bachelor"s degree in information technology (with first class Hons.) from Deakin University, Geelong, VIC., Australia, in 2012. He is currently working toward the Ph.D degree in the School of Software and Electrical Engineering at the Swinburne university of technology. His research interest is the application of deep learning techniques for software vulnerability detection.



Jun Zhang received the Ph.D. degree in computer science from the University of Wollongong, Wollongong, Australia, in 2011. He is an Associate Professor with the School of Software and Electrical Engineering and the Deputy Director of Swinburne Cybersecurity Lab, Swinburne University of Technology, Australia. He has published over 90 research papers in refereed international journals and conferences, such as the IEEE COMMUNICATIONS SURVEYS AND TUTORIALS, the IEEE/ACM TRANSACTIONS ON NETWORKING, the IEEE TRANSACTIONS ON IMAGE PROCESSING, the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, the IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY, the ACM Conference on Computer and Communications Security, and the ACM Asia Conference on Computer and Communications Security. His research interests include cybersecurity and applied machine learning. He has been internationally recognized as an Active Researcher in cybersecurity, evidenced by his chairing (PC Chair, Workshop Chair, or Publicity Chair) of eight international conferences since 2013, and presenting of invited keynote addresses in two conferences and an invited lecture in IEEE SMC Victorian Chapter.



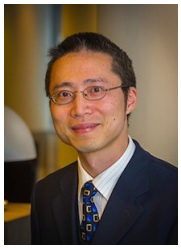
Wei Luo is a Senior Lecturer in Data Science at Deakin University. Wei's recent research focuses on machine learning and its application in health, sports, and cybersecurity. Wei has tackled a number of key information challenges in healthcare delivery and has published more than 50 papers in peer-reviewed journals and conferences. Wei holds a PhD in computer science from Simon Fraser University, where he received training in statistics, machine learning, computational logic, and modern software development.

Lei Pan (M'12) received the Ph.D. degree in computer forensics from Deakin University, Australia, in 2008. He is currently a Senior Lecturer with the School of Information Technology, Deakin University. His research interests are cyber security and privacy. He has authored 50 research papers in refereed international journals and conferences, such as the IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY, the IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, and the IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS.

Olivier De Vel Dr Olivier de Vel obtained a PhD in Electronic Engineering from the Institut National Polytechnique of Grenoble (INPG, France). He is currently Principal Scientist (Cyber) in the Cyber and Electronic Warfare Division, Defence Science and Technology (DST) Group, Department of Defence, Australia.

Dr de Vel has worked at several national and international universities, in government research agencies, and in industry R&D laboratories. Dr de Vel joined DST Group in 1999 to set up and provide the scientific R&D leadership in cyber forensics. In 2005, he was appointed Research Leader in Cyber Assurance and Operations to lead the DST Group broad spectrum cyber-security program in the C3I Division. Dr de Vel's expertise is in the area of artificial intelligence and machine learning for cyber-security and he has published over 100 papers in computer science, digital forensics and machine learning.

Paul Montague obtained a PhD in Mathematical Physics from the University of Cambridge. Following a research career in Mathematical Physics, he transitioned into computer security and developed security and cryptographic solutions for Motorola. He now pursues research in computer security in the Defence Science and Technology Group of Australian Defence. His research interests currently include the application of machine learning and data analytic techniques to cyber security problems.



Yang Xiang received the Ph.D. degree in computer science from Deakin University, Australia. He is currently a Full Professor and the Dean of Digital Research and Innovation Capability Platform, Swinburne University of Technology, Australia. His research interests include cyber security, which covers network and system security, data analytics, distributed systems, and networking. In particular, he is currently leading his team developing active defense systems against large-scale distributed network attacks. He is the Chief Investigator of several projects in network and system security, funded by the Australian Research Council. He has published over 200 research papers in many international journals and conferences. He served as an Associate Editor for the IEEE TRANSACTIONS ON COMPUTERS, the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, and Security and Communication Networks (Wiley), and an Editor for the JOURNAL OF NETWORK AND COMPUTER APPLICATIONS. He is the Coordinator (Asia) for IEEE Computer Society Technical Committee on Distributed Processing.