DeepBalance: Deep-Learning and Fuzzy Oversampling for Vulnerability Detection

Shigang Liu, *Member, IEEE,* Guanjun Lin, Qing-Long Han, *Fellow, IEEE,* Sheng Wen, *Member, IEEE,* Jun Zhang, *Senior Member, IEEE,* and Yang Xiang, *Fellow, IEEE*

***Abstract*—Software vulnerability has long been an important but critical research issue in cybersecurity. Recently, the machine learning (ML) based approach has attracted increasing interest in the research of software vulnerability detection. However, the detection performance of existing ML-based methods require further improvement. There are two challenges: one is code representation for machine learning and the other is class imbalance between vulnerable code and non-vulnerable code. To overcome these challenges, this paper develops a DeepBalance system, which combines the new ideas of deep code representation learning and fuzzy-based class rebalancing. We design a deep neural network with Bidirectional Long Short-Term Memory (BiLSTM) to learn invariant and discriminative code representations from labelled vulnerable and non-vulnerable code. Then, a new fuzzy oversampling method is employed to rebalance the training data by generating synthetic samples for the class of vulnerable code. To evaluate the performance of the new system, we carry out a series of experiments in a real-world ground-truth data set that consists of the code from the projects of LibTIFF, LibPNG, and FFmpeg. The results show that the proposed new system can significantly improve the vulnerability detection performance. For example, the improvement achieves 15% in terms of F-measure.**

***Index Terms*—Software vulnerability detection, class imbalance, machine learning, deep learning, feature learning.**

I. Introduction

WHEN digital services and software products have become ubiquitous in people's life [1], vulnerability detection that certifies the security of using the software has also become an important research area in both academia and industries [2], [3]. In the context of software security, vulnerability refers to a specific flaw or oversight in a piece of software that allows attackers to compromise the system. So far, the number of vulnerabilities has been reported to be positively correlated to the volume of software copies. For example, the well-known CVE (Common Vulnerabilities and Exposures) database only registered around 4500 vulnerabilities in 2010 and the number increased to over 8000 in 2014 [4]. Until 2017, the record has explosively risen to 17265 vulnerabilities in the CVE database. In another example, more than 43000 software vulnerabilities have been collected by NVD (National Vulnerability Database) since 1997 [5]. These vulnerabilities affected the secure usage of more than 17000 digital services/products (e.g. end-user software, and device firmware), causing direct financial losses of 226 billion dollars a year [6]. Recent studies suggested that when more digital services/products are developed in the software market, the increase of software vulnerabilities will continue in the years to come [7], [8].

To overcome this challenge, many machine learning(ML)-based methods have been proposed to detect software vulnerabilities [4], [9], [10], [11], [12]. These ML-based methods firstly train the classifiers using the labelled training dataset that consists of vulnerable and non-vulnerable samples, and then the trained classifiers are used to identify the unknown vulnerable samples in the testing data sets according to a sort of "distance" to the existing vulnerability features. However, recent studies have found that current ML-based methods are suffering from the critical "Class Imbalance" problem [9], [13], [14], [15], [16]. This problem refers to the uneven distribution of class samples in the data sets. As ML-based classifiers generally bias to the classes that take the majority of samples in the data sets, the "Class Imbalance" problem largely reduce the accuracy of identifying the samples that suggest vulnerabilities but usually rarely appear in the data sets. In fact, the class samples that usually indicate vulnerabilities are naturally rare in the real data sets. Therefore, we are strongly motivated to propose a new method in this paper, which can address the "class imbalance" problem [17].

In this paper, we first develop a deep learning(DL)-based scheme for high-level feature representation learning based on the source code we collected. Then, a fuzzy-based oversampling for re-balancing the ground-truth training data will be proposed. Specifically, we overcome the difficulty of automatically extracting deep vulnerable programming features which are expected to contain much information than the shallow features driven by domain knowledge. Our approach assumes that vulnerable programming patterns are associated with many potential vulnerabilities, and these patterns can be disclosed by analysing the program's Abstract Syntax Trees (ASTs). Considering there is class imbalance problem in the ground truth data set, we present a fuzzy-based oversampling (FOS) method to alleviate the imbalance problem. FOS creates synthetic vulnerable class samples in order to 're-balance' the uneven distribution between vulnerable samples and non-vulnerable samples. We summarise the major contributions as follows:

- We proposed a DL-based method that using bi-directional Long Short-Term Memory (LSTM) networks for learning high-level representations of ASTs. The proposed method addressed the problem of extracting features from the cross-project source code we collected.

S. Liu, G. Lin, Q.L. Han, S. Wen, J. Zhang, and Y. Xiang are with School of Software and Electrical Engineering, Swinburne University of Technology, Hawthorn VIC 3122, Australia, e-mail: ({shigangliu, glin, qhan, swen, junzhang, yxiang}@swin.edu.au).Corresponding author: Jun Zhang.

- We also proposed an innovative fuzzy-based oversampling scheme that can use existing data to create synthetic samples in order to redistribute the imbalanced data.
- We carried out a series of empirical and theoretical studies based on real-world data. The results demonstrated the outperformance of our DeepBalance system in software vulnerability detection. As far as we know, we are among the first to fundamentally study the class imbalance problem in software vulnerability detection when deep learning is applied.
- We published the ASTs data set for other researchers to examine and contribute to our proposed system. The data set can be downloaded from *https://github.com/DanielLin1986/function_representation_learning*.

The rest of the paper is organised as follows. Section III discussed the class imbalance problem. Section IV presents the details of the proposed method. The experiments study and results are described in Section V. Section II reviews the related work, and finally Section VI concludes this paper.

II. Related Work

We first recall the related work about machine learning-based techniques for software vulnerability detection [15], and then we review the class imbalance related work.

A. Related work for software vulnerability detection

Many studies have been proposed to prevent vulnerabilities. Among these approaches, machine learning-based techniques are playing an important role in software vulnerability detection [18], [19]. Morrison et al. [14] demonstrated that the recall and precision performance of software vulnerability detection based on binary-level prediction outperforms file-level prediction. Shar and Tan [20] propose to use static code attributes for software vulnerability detection. The best experimental results using C4.5, NB [2] and MLP achieved an averaged results of 93% recall and 11% false alarm rate in detecting SQL injection (SQLI) vulnerabilities. Yamaguchi et al. [21] developed a scheme that can automatically infer search patterns for taint-style vulnerabilities from C source code. Alves et al. [6] evaluated several states of the art vulnerability prediction techniques using a large and representative data set. Results showed that random forest achieves very high results in all of the metrics. Eschweiler et al. [22] emphasised the importance of security-critical vulnerabilities detection over binary code, and presented a new approach to efficiently search for similar functions based on binary code. Grieco et al. [23] developed a machine learning-based approach that uses lightweight static and dynamic features for memory corruption vulnerability analysis over binary code.

Component-level vulnerability detection is coarse-grained. Previous work [24], [25] have proposed file-level vulnerability prediction models relying on the feature sets derived from measures such as code complexity metrics, code changes and prior-identified software faults. A finer-grained approach was presented by Yamaguchi et al. [26] for extrapolating vulnerable functions. They applied Principal Component Analysis (PCA) for composing of API usage patterns of functions.

There are prior studies working on cross-project fault detection [27]. Nam et al. [28] applied a transfer learning technique called transfer component analysis (TCA) [29] for cross-project fault prediction at the file-level. Wang et al. [30] leveraged deep belief networks (DBN) to extract semantic features from ASTs that are derived from Java source code. However, their detection ran at the file-level. Our work that differs from Wang et al. [30] is that we not only apply Bi-directional LSTM networks for learning representations at function-level but also focus on finer-grained detection, capable of pinpointing vulnerable functions.

Apart from this, feature extraction processes mainly rely on heuristics, skills and experience of knowledgeable individuals. The choice of features usually not only affects the detection capability but also has a negative impact on the detection granularity. Neuhaus et al. [31] discovered that vulnerable software components shared similar sets imports and function calls. Scandariato et al [32] employed bag-of-words (BoW) for feature representations, in which a software component is seen as a series of terms with associated frequencies. Hoa et al. [33] developed a novel deep learning-based approach that can automatically learn features for predicting vulnerabilities in software code. Metrics derived from developer activity were firstly utilized by Pinzger et al. [34] for inferring the likelihood of post-release failures in software modules. The authors discovered that metrics such as the number of contributors of a file and the number of commits could be important predictors of failures in software. Later, Meneely and Williams [35] linked developer activity to software security. Their study showed that some of the features are selected by knowledgeable domain experts which may carry outdated experience and underlying biases.

B. Related work for imbalance data learning

In the last decades, a large number of works have been reported to tackle the problem of imbalanced data [14], [15], [36], which can be categorized into three categories: cost-sensitive learning and ensemble learning such as RAMO [37]. In this subsection, we specifically recall the oversampling techniques, because they are closely related to our proposed technique. For detailed reviews, please refer to [38].

The data sampling approach is actually a re-balancing process for an imbalanced data set. One of the most common practice is oversampling the minority class. Oversampling creates synthetic minority samples by either randomly appending replicated samples or 'intelligently' generating samples [38]. He et al. developed an Adaptive Synthetic Sampling Technique (ADASYN) for generating synthetic samples [39]. Later, a cluster-based oversampling (CBOS) was proposed to even out the between-class imbalance as well as the within-class imbalance [40]. Barua proposed a Majority Weighed Minority (MWMOTE) oversampling technique [41]. Hao et al. developed the GLMBoost algorithm coupled with SMOTE [42] to address the imbalance problem, particularly for several imbalanced data sets from PubChem BioAssay [43]. Zhao

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/TFUZZ.2019.2958558, IEEE Transactions on Fuzzy Systems

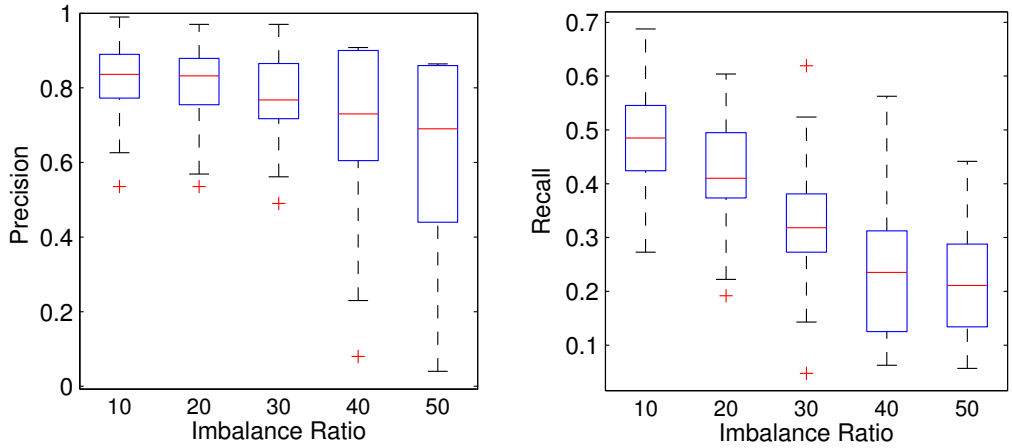


Fig. 1: The degradation of Precision and Recall caused by the class imbalance problem.

Figure Notes: This figure shows the Precision and Recall performance results. We can see the medium Precision values witness a significant decrease with the increase of the IR. The medium precision value is about 85% with IR equals 10, however, this values down to less than 70% when IR is 50. In the meanwhile, the medium Recall values drops from about 47% to about 21% with the IR increased from 10 to 50.

et al. proposed a stratified over-sampling method to generate diverse balanced training sets for random forests by clustering the original training data set multiple times [44]. In [45], a new simple experimental design has been conducted regarding the class imbalance learning algorithms, and the experimental results show that ROS outperforms some sophisticated methods. In [16], a fuzzy-based information decomposition (FID) method has been proposed to address the class imbalance problem with missing values presented. Grieco et al. employed random oversampling (ROS) to redistribute the imbalanced training data in software vulnerability detection [23]. Lin et al. simply applied cost-sensitive learning by assigning more weight to the vulnerable (minority) class [19]. Ban et al. [46] conducted an experimental study and demonstrated that class imbalance problem has a negative impact on software vulnerability detection. Therefore, the class imbalance problem should be taken into consideration in software vulnerability detection.

In addition, there is also not so much work that using deep learning for software vulnerability detection, the most closely work that related to this study are the work presented by Lin et al. [47] (we refer the algorithm as G-VulD in this paper), Dam et al. [33] (refer as LSTM-FL in this paper), and Li et al. [48] (refer to VulDeePecker in this paper). G-VulD employs deep learning for function-level vulnerability discovery in terms of unlabelled data; LSTM-FL employs LSTM to learn important information from the long context. It leverages LSTM to capture the long context relationships in the source code where dependent code elements are scattered far apart. While VulDeePecker is a deep learning based system for software vulnerability detection. However, the aforementioned studies are not designed for the class imbalance problem. As far as we know, we are the first to develop a fuzzy-based oversampling for dealing with the class imbalance problem in software vulnerability detection. We believe our work is fundamentally important in the area of software vulnerability detection.

III. Problem Statement

In real-world ML-based applications, the "class imbalance" problem can reduce the effectiveness of their data mining and machine learning models, particularly in predicting minority classes. This means that the classification model usually simply classifies the majority classes in every case and becomes increasingly trivial to achieve high prediction accuracy. In general, conventional classifiers fail to properly learn the distributive characteristics of imbalanced data sets since they ignore the minority class instances, treating them as noise data. A previous study [14] showed that class imbalance has a negative impact on software vulnerability detection. For example, the experimental results are even worse when conducted on source code-level with the percentage of vulnerability being 0.33%. The Recall is less than 14% while the Precision is less than 47%. Therefore, the "class imbalance" problem should be addressed when applying machine learning algorithms to software vulnerability detection.

In order to examine the effect of the "class imbalance" problem to software vulnerability detection, we conduct a series of preliminary experiments based on the imbalanced data varying from 10 to 50 with incremental steps of 10 (i.e., IR = 10, 20, 30, 40, 50).

$$\text{IR} = \frac{\text{\# majority class samples}}{\text{\# minority class samples}} \tag{1}$$

Fig.1 demonstrates the Precision and Recall performance results. One can see that the medium Precision values witness a significant decrease with the increase of the IR. The medium precision value is about 85% with IR equals 10. However, this value decreases to less than 70% when IR is 50 (the results are based on the averaged 10 runs (of 10-fold cross validation) of the experiments among 5 independent data sets). Meanwhile, the medium Recall value (i.e., TPR: true positive rate) drops from about 47% to about 21% with the IR increasing from 10 to 50. This can be explained by the fact that with the increased number of not vulnerable functions, the classification algorithm failed to build a robust model that can capture

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/TFUZZ.2019.2958558, IEEE Transactions on Fuzzy Systems

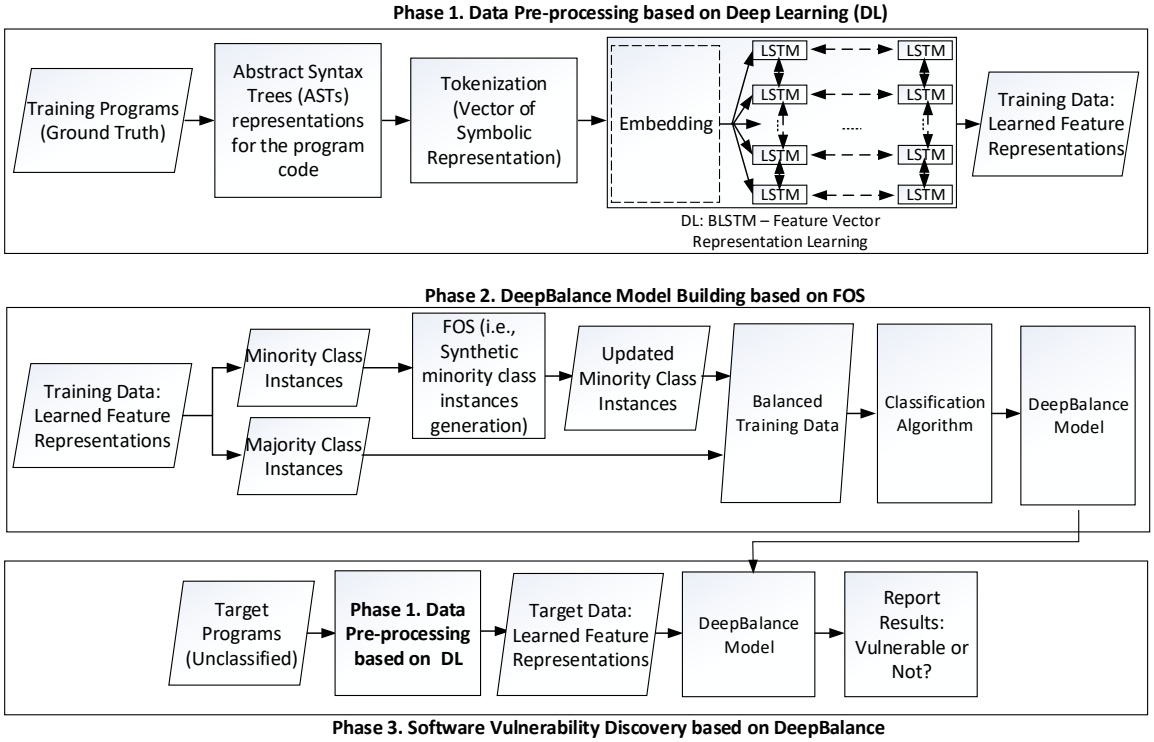


Fig. 2: The proposed DeepBalance framework.

the vulnerable functions, which means the classifier is biased toward the not vulnerable samples (majority class) due to the "class imbalance" problem. Therefore, it is very necessary to deal with the class imbalance problem when using machine learning algorithms for software vulnerabilities detection.

IV. Methodology

In this section, we present the main idea of the proposed DeepBalance, which is an ML-based approach for software vulnerability discovery with class imbalance problem being solved. Fig.2 describes the proposed framework.

As shown in Fig.2, there are three stages in the proposed algorithm: 1) Data pre-processing based on deep learning (DL); 2) DeepBalance model building based on fuzzy over-sampling (FOS); 3) Software vulnerability discovery based on DeepBalance. In the first phase, the inputs are source code functions with labels. Each function is labelled as vulnerable or non-vulnerable. Then, the functions are converted to ASTs in a serialized format. Before the serialized ASTs are fed to the network, they are tokenized to form the AST sequences. Subsequently, the Bidirectional LSTM (Bi-LSTM) is used to learn the feature vector representations of the AST sequences. In the second phase, FOS is developed to deal with the class imbalance problem when facing the imbalanced data in which the number of non-vulnerable samples is outnumbered of the number of vulnerable samples. Then, Random Forest is employed to build a classification model. The new model is named as DeepBalance in Phase 2. The input to the third phase is the source code functions of the target program. We apply the same data processing methods to covert the source code functions to tokenized AST sequences and feed them to the Bi-LSTM network. The output is whether a function is vulnerable or not based on the confidence produced by the classifier.

In the following discussion, we will first explain the Bi-LSTM networks used for learning representations at the function-level. We will then discuss the proposed fuzzy-based oversampling scheme used for redistributing the training data.

A. Data processing

Our experiments involve three open-source projects: LibTIFF, LibPNG, and FFmpeg. These projects are popular libraries used in the open-source community. We obtain the source code of these projects from Github. Specifically, we manually labeled 329 vulnerable functions and collected 6,756 non-vulnerable functions from the three open-source projects. The labels of vulnerable functions were obtained from the National Vulnerability Database (NVD) and the Common Vulnerability and Exposures (CVE). Based on the description of the two vulnerabilities databases, we downloaded the corresponding versions of the source code of each project and locate each vulnerable function in the source code and labeled it accordingly. In this paper, we consider the function-level

```
1 int foo (int a) {
2     int c = 0;
3     /* Invoke function bar() */
4     int b = bar (a);
5         if (a == 10) {
6             c = a + b;
7         }
8     return c;
9 }
```

# Type	Depth	Value 1	Value 2
func	0	int	foo
params	1		
param	2	int	a
stmnts	1		
decl	2	int	c
op	2	=	
decl	2	int	b
op	2	=	
call	3	bar	
arg	4	a	
if	2	(a == 10)	
cond	3	a == 10	
op	4	==	
stmts	3		
op	4	=	
op	4	+	
return	2	c	

Fig. 3: An example of C function (right) and its AST in a serialized format (left).

vulnerabilities. For vulnerabilities that span multiple functions or multiple files, we discard them (i.e., the vulnerable functions and the vulnerabilities which span across multiple functions and/or files are not included).

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

1) AST-based Feature Extraction: The source code of three open source projects is used to obtain the ASTs. Parsing ASTs from source code is non-trivial since a build environment should be present for code compilation. With "*CodeSensor*[1]", which is a robust parser implemented by [49] based on the concept of *island grammars* [50], we could acquire ASTs from source code functions without supporting libraries and dependencies. The input of *CodeSensor* is the program's source code, and the output is all the functions' AST in a serialized format as Fig.3 shows. Specifically, the serialized AST format uses a table to list the tree structure. The *Depth* column marks the position of the components of the source code function. For example, in the first row of the table (the left part of Fig. 3), the *Type* name *func* and the *Depth 0* means that function name *foo* and the return value *int* is the root of the tree. The second row *params* means the function takes at least one parameter, and the *Depth 1* means it is the first child of the root node and so forth.

With parsed ASTs, we aim at converting them to vectors while preserving the structural information. To achieve this, we use depth-first traversal (DFT) to map the nodes of ASTs to elements of vectors so that each node will become an element of a vector. The sequence of elements in the vector matters because they partially reflect the hierarchical position of the nodes in an AST. By using the DFT to convert an AST to a textual vector, the root node *foo* will be the first element of the vector which is the name of a function, and the second element *int* denotes the return type of the function *foo*. The third element *params* means that the function *foo* takes parameter(s). The fourth and the fifth elements will be *params* and *int*, which specify the function *foo* takes a parameter of an integer. After mapping the AST of a function to a textual vector, it will be in a form like: [*foo, int, params, param, int, a, stmnts, decl, int, c, op, =, decl, ...*]. This textual vector is a special semantic sentence for the function foo. The semantic meaning is held by each element of the vector and their sequence.

Next, we need to tokenize textual elements of the AST nodes to convert the textual vector to a numeric one. A mapping is built to link each textual element of vectors to an integer. These integers act as "tokens" which uniquely identify each textual element. For instance, type "int" is mapped to "1", keyword "static" is mapped to "2", and so on. By doing this, each element within the vectors was replaced with numeric tokens and their sequence remains intact. To consider the code semantic meanings, Word2Vec [51] is applied to convert each element of the vector to meaningful embeddings. We train a Word2Vec model using the continuous bag-of-words (CBOW) architecture with default settings provided by the Gensim package. The unsupervised training process does not require any labelled data. We use the code base of three open source projects to train the Word2Vec model. The outcome of training is a dictionary containing mappings between each word (code element) and its corresponding 100-dimensional vector embedding.

2) LSTM Network for AST-based Representations: We observe an interesting resemblance that exists between ASTs and sentences in natural languages. As we discussed in the previous paragraph, the sequence of the elements in a vector partially represents the structure of the ASTs and forms the "*function meanings*". Furthermore, there are connections among elements, which forms the contextual information. For example, the first element of a vector is the name of a function, followed closely by the functions return type. The element immediately after function name "*foo*" should be its return type such as "*int*" or "*void*" and should not be other C/C++ keywords such as "*decl*" or "*return*". The LSTM architecture is built to handle sequential data and learn contextual dependence.

Another reason which supports the idea of applying LSTM is that the vulnerable programming patterns in a function can usually be associated with more than one lines of code. When we map functions to vectors, patterns linked to vulnerabilities are related to multiple elements of the vector. The standard RNNs are capable of handling short-term dependencies such as the element "*foo*" which should be followed by type name "*int*" or "*void*", but they have problem of dealing with long-term dependencies such as capturing the vulnerable programming patterns that are related to many continuous or intermittent elements. Consequently, we use RNN with LSTM cells to capture long-term dependencies for learning high-level representations of potential vulnerabilities. Moreover, identifying vulnerable programming patterns also requires the consideration of both the preceding and succeeding context.

[1]https://github.com/fabsx00/codesensor

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/TFUZZ.2019.2958558, IEEE Transactions on Fuzzy Systems

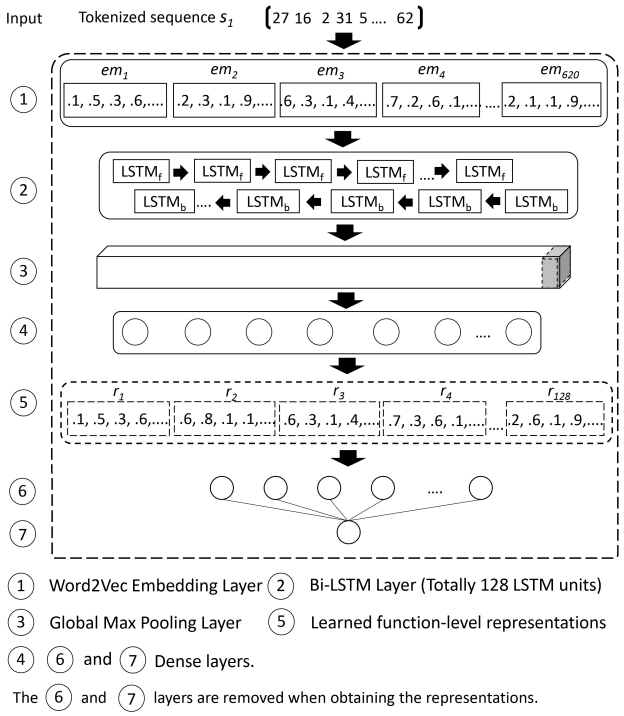


Fig. 4: The Bi-LSTM network for function representation learning.

Therefore, we apply the bidirectional form of the LSTM network for learning the contextual dependencies.

To acquire the unified length of input vectors, we apply padding and truncation to handle the vectors of various length. Considering the balance between the length and over sparsity of vectors, we choose 620 as the padding/truncation threshold based on the observation that 85% of our converted AST sequences contain less than 620 elements. We keep the first 620 elements of long sequences and use zeros for end-of-sequence padding for short sequences. The network takes a mini-batch of converted AST sequences as input and predicts the vulnerability probabilities. "1" indicates a vulnerable function and "0" indicates a non-vulnerable function. In our experiments, we set the batch size to 16 and the epoch number to 150, which yields the best performance.

The architecture of our proposed LSTM-based network contains 6 layers, as Fig. 4 shows. The design of the network architecture is based on the nature of code and the practice of vulnerability discovery. The settings of the network are tuned according to the performance optimization on our initial experiments. The first layer is a Word2Vec embedding layer which maps each element of the sequence to a meaningful dense vector of fixed dimensionality. With embedding, discrete inputs such as the code components are converted to dense vectors. The second layer is an LSTM layer which contains 64 LSTM units in a bi-directional form (altogether 128 LSTM units). The bi-directional LSTM layer is designed to learn the dependencies of each element for generating higher-level abstractions. Then, a global max pooling layer is followed by the bi-direction LSTM layer for strengthening the signals. Specifically, since we label the vulnerable functions as "1", we assume that the representation of vulnerable functions would result in large values which are close to 1. The global max pooling layer automatically picks up the maximal values, which helps to select potentially vulnerable representations. The last three layers are dense layers for converging a 128-dimensionality output to a single dimensionality. To prevent overfitting, we apply dropout with the value of 0.5 between the third and second last layers. During the training phase, the loss function that we attempt to minimize in our experiments is a binary cross entropy function, and the optimizer used is the "Adamax". More detailed settings of our proposed network are provided in Table I.

TABLE I: The settings of proposed network.

Layer	Activation Function	Output Shape	Param #
Embedding	None	(batch_size, 620, 100)	3,481,100
Bi-LSTM	"tanh"	(batch_size, 620, 128)	84,992
Golbal Max Pooling	None	(batch_size, 128)	0
Dense 1	"tanh"	(batch_size, 128)	16,512
Dense 2	None	(batch_size, 64)	8,256
Dense 3	"sigmoid"	(batch_size, 1)	65

Prior to acquiring the learned representations, it needs to evaluate whether our new model is capable of offering acceptable detection performance. We train the neural network model with the training set that consists of the labelled vulnerable and non-vulnerable functions in serialized AST format. The network is tuned to maximize the performance during training. Once the model is trained, we feed the trained network with both the training and the test sets of the data and obtain the learned representations from the fourth layer of the network (the last two layers are removed). Given a sequence as the input, the learned representation is a 128-dimensional vector. These representations will be used in the subsequent task of training a random forest classifier.

B. FOS: Fuzzy-based Oversampling

There are three main phases involved in FOS. The first phase (discussed in IV-B1) partitions the confidence interval (CI) into small intervals, $A_i, i = 1, 2, \cdots, t$ (where t is the number of values to be generated), which will be used for calculating the membership degree of observations in each interval. The second phase calculates the contribution of the observed data in each small interval obtained from the first phase (described in subsection IV-B2), $\mu_{A_i}(x_j)$, where in $\mathbf{x} = (x_1, x_2, \cdots, x_n)^T$ denotes the observed data in $\mathbf{x}$. The final phase generates the synthetic data (illustrated in IV-B3), $\tilde{m}_i$, using the calculated contributions of the observations. The complete method is presented in [Algorithm 1].

1) Construction of Small Intervals: Let's start with $\mathbf{x} = (x_1, x_2, \cdots, x_m)^T$ to be a feature vector that is composed of m values. Actually, $\mathbf{x}$ can be any column feature vector from the minority (vulnerable) class. After each of the feature vector follows the same process, the imbalanced data will be re-distributed. We set the number of synthetic samples to be generated as t.

Denote standard deviation of $\mathbf{x}$ as σ. There are two scenarios when computing the confidence intervals in the real-world scenario: 1) σ is known, and 2) σ is unknown. In this subsection,

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/TFUZZ.2019.2958558, IEEE Transactions on Fuzzy Systems

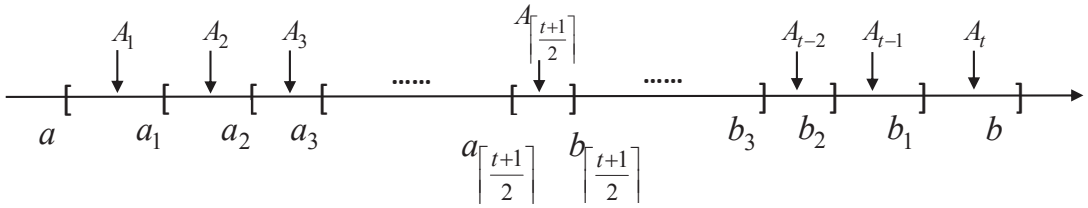


Fig. 5: Illustration of interval A_i obtained from CI.

Algorithm 1 FOS Algorithm

```
INPUT: oversampling rate α, minority class samples X_min
OUTPUT: over-sampling minority class data set with extra synthetic samples created: X̃_min;
for each feature vector x in X_min or X_maj; do
    identify the number, t, of data, to be generated for x, t = number of minority samples ×α ;
    calculate t small intervals A_i based on 100×(1−α_i)% CI;
    for i = 1 : t do
        compute the contribution of the observed data x_j in A_i, μ_{A_i}(x_j);
        compute the generated data, m̃_i, for A_i;
    end for
end for
return X̃_min;
```

we begin by providing a way to calculate a $100 \times (1-\alpha)\%$ CI for the mean, in terms of population standard deviation, σ is unknown. This is because in practice, it is hard to know the σ value of the data set.

According to Central Limit Theorem (CLT) [52] we consider the population standard deviation σ as unknown, as it will usually be in practice and has to be estimated by the sample standard deviation s. We can calculate the $(1-\alpha) \times 100\%$ CI for the mean μ of any arbitrary random variable $\mathbf{x}$ in terms of sample size $\geqslant 30$ as

$$\left[\overline{\mathbf{x}} - \mathbf{z}_{\frac{\alpha}{2}} \frac{\mathbf{S_x}}{\sqrt{\mathbf{n}}}, \overline{\mathbf{x}} + \mathbf{z}_{\frac{\alpha}{2}} \frac{\mathbf{S_x}}{\sqrt{\mathbf{n}}}\right] \tag{2}$$

where $z_{\frac{\alpha}{2}}$ is the $1 - \frac{\alpha}{2}$ percentile of $N(0,1)$ that the area to its right at a significance level of α.

$$\overline{\mathbf{x}} = \frac{x_1 + x_2 + \cdots + x_m}{m}$$

$$S_{\mathbf{x}} = \sqrt{\frac{\sum_{j=1}^{m} (x_j - \overline{\mathbf{x}})^2}{m-1}}.$$

And a $(1-\alpha) \times 100\%$ CI for the mean μ of any arbitrary random variable $\mathbf{x}$ in terms of sample size < 30 is given as:

$$\left[\overline{\mathbf{x}} - \mathbf{t}_{\frac{\alpha}{2}} \frac{\mathbf{S_x}}{\sqrt{\mathbf{n}}}, \overline{\mathbf{x}} + \mathbf{t}_{\frac{\alpha}{2}} \frac{\mathbf{S_x}}{\sqrt{\mathbf{n}}}\right] \tag{3}$$

wherein $t_{\frac{\alpha}{2}}$ denotes the $\frac{\alpha}{2}$ percent of the student's t-distribution. When the size of the data set is small and the population standard deviation σ is unknown, t-based CI will correctly reflect the added uncertainty. Therefore, in this paper, we will employ $t_{\frac{\alpha}{2}}$ instead of $z_{\frac{\alpha}{2}}$. In the following part, we will specifically pick up the scenario of $m \geqslant 30$ for description. In the scenario of $m < 30$, the same process can be applied.

Denotes $[a_i, b_i] = \left[\overline{\mathbf{x}} - \mathbf{z}_{\frac{\alpha_i}{2}} \frac{\mathbf{S_x}}{\sqrt{\mathbf{n}}}, \overline{\mathbf{x}} + \mathbf{z}_{\frac{\alpha_i}{2}} \frac{\mathbf{S_x}}{\sqrt{\mathbf{n}}}\right]$ for the $100 \times (1-\alpha_i)\%$ CI of $\mathbf{x}$. α_i is given as,

$$\alpha_i = \left(\frac{100}{\lceil \frac{t+1}{2} \rceil} \times i\right), \quad i = 1, 2, \cdots, \left\lceil \frac{t}{2} \right\rceil$$

wherein $\lceil\ \rceil$ means take the upper integer bound.

For each α_i, we immediately obtain the $(1-\alpha_i) \times 100\%$ CI, $[a_i, b_i]$, via formula 2. Accordingly, t intervals such as $A_1, A_2, \cdots, A_t$ can be given from Fig.5. That is:

$$\begin{cases} A_1 = [a, a_1] \\ A_2 = [a_1, a_2] \\ \quad \vdots \\ A_{\lceil \frac{t+1}{2} \rceil} = [a_{\lceil \frac{t+1}{2} \rceil}, b_{\lceil \frac{t+1}{2} \rceil}] \\ \quad \vdots \\ A_{t-1} = [b_2, b_1] \\ A_t = [b_1, b] \end{cases}$$

wherein

$$\begin{cases} a = min\{x_i | i = 1, 2, \cdots, m\} \\ b = max\{x_i | i = 1, 2, \cdots, m\}. \end{cases}$$

2) Calculation of Contribution: In the last subsection IV-B1, we discussed how to obtain the intervals. Obviously, the intervals we obtained are all real intervals. Therefore, they are all Lebesgue measurable. In this case, the length of any interval can be calculated.

For $i = 1, 2, \cdots, t$, h_i denotes as the length of the interval A_i, and u_i as the centre of A_i. For example, if $A_{\lceil \frac{t}{2} \rceil} = [a_{\lceil \frac{t}{2} \rceil - 1}, b_{\lceil \frac{t}{2} \rceil - 1}]$, then

$$\begin{cases} h_{\lceil \frac{t}{2} \rceil} = b_{\lceil \frac{t}{2} \rceil - 1} - a_{\lceil \frac{t}{2} \rceil - 1} \\ u_{\lceil \frac{t}{2} \rceil} = \frac{a_{\lceil \frac{t}{2} \rceil - 1} + b_{\lceil \frac{t}{2} \rceil - 1}}{2}. \end{cases}$$

The success of the FOS algorithm is largely dependent on how we make use of the observed data to generate new data from the column feature vector. For this purpose, FOS uses the fuzzy membership function (MF), which is also called [0, 1]-valued membership function, to identify the membership degree of $\mathbf{x}_j$ in A_i. In fuzzy theory, the value of membership degree ranges from '0' to '1'. '0' means that $\mathbf{x}_j$ is not

a member of A_i, and the value '1' means that $\mathbf{x}_j$ is a full member of A_i, while the values between '0' and '1' characterise the degree of $\mathbf{x}_j$ that belongs to A_i. In this work, we compute the membership degree of $\mathbf{x}_j$ in A_i as follows:

$$\mu_{A_i}(x_j) = \begin{cases} 1 - \frac{\|x_j - u_i\|}{h_i} & \text{if } \|x_j - u_i\| \leq h_i \\ 0 & \text{if } \|x_j - u_i\| > h_i \end{cases} \quad (4)$$

wherein u_i is the centre of A_i. And the membership degree $\mu_{A_i}(x_j)$ is defined as the contribution of $\mathbf{x}_j$ in A_i.

In our scenario, the larger value in $\mu_{A_i}(x_j)$ means more contributions of x_j in A_i. Specifically, $\mu_{A_i}(x_j) = 0$ means x_j has no contribution in A_i, while $\mu_{A_i}(x_j) = 1$ means the full contribution of x_j for A_i.

Moreover, according to formula 4, we can see that the observations that close to u_i have larger contributions. In the special situation where $x_j = u_i$, we can obtain $\mu_{A_i}(x_j) = 1$.

3) Data Generation: We have partitioned each $100 \times (1-\alpha)\%$ CI of the observed data into t small intervals. Theoretically speaking, each small interval A_i contains the same number of values. Based on formula 4, regarding the contribution of $\mathbf{x} = (x_1, x_2, \cdots, x_m)^T$ in A_i, we will have

$$\mu_{A_i}(\mathbf{x}) = (\mu_{A_i}(x_1), \mu_{A_i}(x_2), \cdots, \mu_{A_i}(x_n)).$$

In order to obtain the generated value $\tilde{m}_i$ from the small interval A_i, we define the mapping

$$F : \mu_{A_i}(\mathbf{x}) \to R$$

Providing $\mu_{A_i}(\mathbf{x})$ to be a non-negative array, the values can be estimated as

$$\tilde{m}_i = F(\mu_{A_i}(\mathbf{x})) = \frac{\sum_{j=1}^{n} \mu_{A_i}(x_j) \times x_j}{\sum_{j=1}^{n} \mu_{A_i}(x_j)} \quad (5)$$

which means we set the weighted average as the value to be generated.

For the case that the contribution of every feature value x_j on interval A_i is 0, which means the weight $\mu_{A_i}(x_j) = 0$, the generated data m_i in the interval A_i is settled as:

$$\tilde{m}_i = F(\mu_{A_i}(\mathbf{x})) = \frac{\sum_{j=1}^{n} x_j}{m} \quad (6)$$

In other words,

$$\tilde{m}_i = \bar{\mathbf{x}},$$

wherein $\bar{\mathbf{x}}$ is the mean of all observed feature values. Finally, $\{\tilde{m}_i | i = 1, 2, ..., t\}$ can be generated for feature vector $\mathbf{x}$.

In formula 5, the contribution $\mu_{A_i}(x_j)$ of each observation x_j is considered when calculating the synthetic values $\tilde{m}_i$. Note that formula 6 is not a particular scenario from formula 5, where $\mu_{A_i}(x_j) \equiv 1$. Instead, it is a specific consideration for

$$\mu_{A_i}(\mathbf{x}) = (\mu_{A_i}(y_1), \mu_{A_i}(y_2), \cdots, \mu_{A_i}(y_n)) = (0, 0, \cdots, 0),$$

V. Experiments and Results Analysis

A. Experimental setup

In order to evaluate the proposed approach, we use a real-world data set we collected in our early study [47], [53]. As described in Section IV-A. We refer to the publicly available data repositories such as National Vulnerability Database (NVD) and the Common Vulnerability and Exposures (CVE) for acquiring the vulnerable functions. The gound-truth data set contains 329 vulnerabilities and 6756 neural functions. In order to examine the impact of the "class imbalance" problem, we construct a series of experimental data sets with ten different class imbalance ratio, which are IR = 10, 20, 30, 40, 50. Take IR = 50 for example, we randomly select 120 vulnerable samples and 6000 neural samples from the original data to form the vulnerable (minority) class and neural (majority) class respectively. Moreover, for each imbalanced data set, we repeat the process for 10 times to derive 10 independent data sets.

In the "class imbalance" problem, the overall classification accuracy itself is not a good metric for measuring the performance of classifiers. In this case, model performance is measured using other commonly used learning metrics including Precision, Recall, and FM (F-measure). In this paper, we consider the minority class (Vulnerable samples) as the positive class and the majority class (Not vulnerable samples) as the negative class.

In the experiments, the 10-fold cross-validation is used to each derived data set. FOS is applied to the training data rather than the whole data set. In each experiment, the results are averaged over 10 runs of 10-fold cross-validation. Random Forest [54] is employed to build a classification model since it has been examined to have the best performance in software vulnerable detection [23].

B. Results Analysis

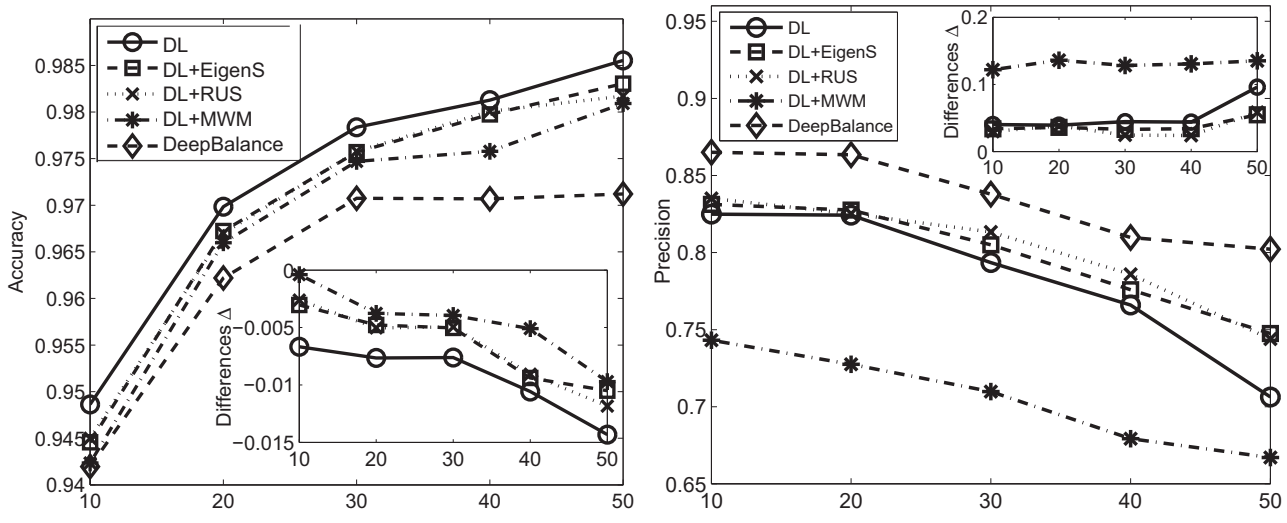
In this study, we mainly focus on answering the following two questions:

Q1: Can DeepBalance outperform other sampling techniques when Deep Learning (DL) is applied for high-quality feature representation learning? Note that, this question is important because DeepBalance employs FOS for dealing with the class imbalance problem in software vulnerability detection, one might interest in the comparison with other sampling techniques. In order to answer this question, some popularly used and recently developed sampling techniques have been considered in this study.

Q2: How effective is DeepBalance when compared with recently developed DL-based software vulnerability detection systems? To answer this question, five recently developed DL-based systems have been compared.

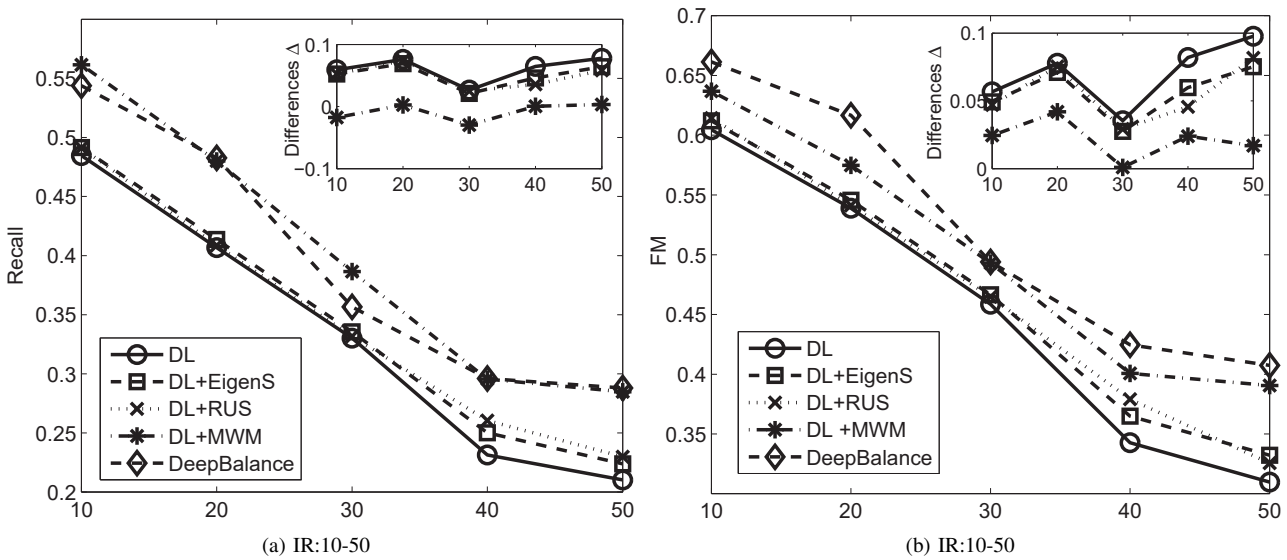
1) Experiments for answering Q1: To answer Q1, given the high quality feature representations learned by DL, we implement MWMOTE (short for MWM) [41], EigenSample [55] and RUS [15] for comparison. We employ MWMOTE (short for MWM) and EigenSample (short for EigenS) techniques not only because they are recently developed algorithms but

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/TFUZZ.2019.2958558, IEEE Transactions on Fuzzy Systems



(a) IR:10-50 (b) IR:10-50

Fig. 6: The Accuracy and Precision values varying IR (Δ denotes the differences between DeepBalance and the other methods). DL performs the best in terms of Accuracy. However, DeepBalance achieves very comparative Accuracy values (with only 0.015 less). DeepBalance can improve the Precision values up to 10% compared to DL.



(a) IR:10-50 (b) IR:10-50

Fig. 7: The Recall and FM values varying IR (Δ denotes the differences between DeepBalance and the other methods). DeepBalance improves the Recall values up to 0.06 compared with DL.

also due to the reason that they perform better than some popularly used algorithms such as SMOTE [42], ADASYN [39], Bootstrapping and RAMO [37]. RUS is considered because it has been employed for handling the class imbalance problem in several software vulnerability studies [15]. According to the previous study, [41] oversampling rate at 200% performs well for most cases. Therefore, we set 200% as the oversampling rate for FOS and MWMOTE. For RUS, an undersampling rate of '50%' is considered in this study. In addition, the original data set without sampling is used to provide a baseline for the performance evaluation (denoted as **DL** in the experimental results).

Fig.6a presents the Accuracy values of the five approaches in terms of IR ranges from 10 to 50. One can see that simple DL performs the best in terms of the Accuracy. However, DeepBalance obtains very comparative Accuracy values, which is only about 0.015 less than DL in general. Fig.6b shows the averaged Precision values of three approaches in terms of IR ranges from 10 to 50. One can see that there are improvements when the imbalanced data have been processed by FOS, RUS and Eigensample methods. However, DeepBalance performs the best in all cases. For example, DeepBalance obtains the best Precision value at about 86% and 80% when IR = 10 and IR = 50, respectively. While

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/TFUZZ.2019.2958558, IEEE Transactions on Fuzzy Systems

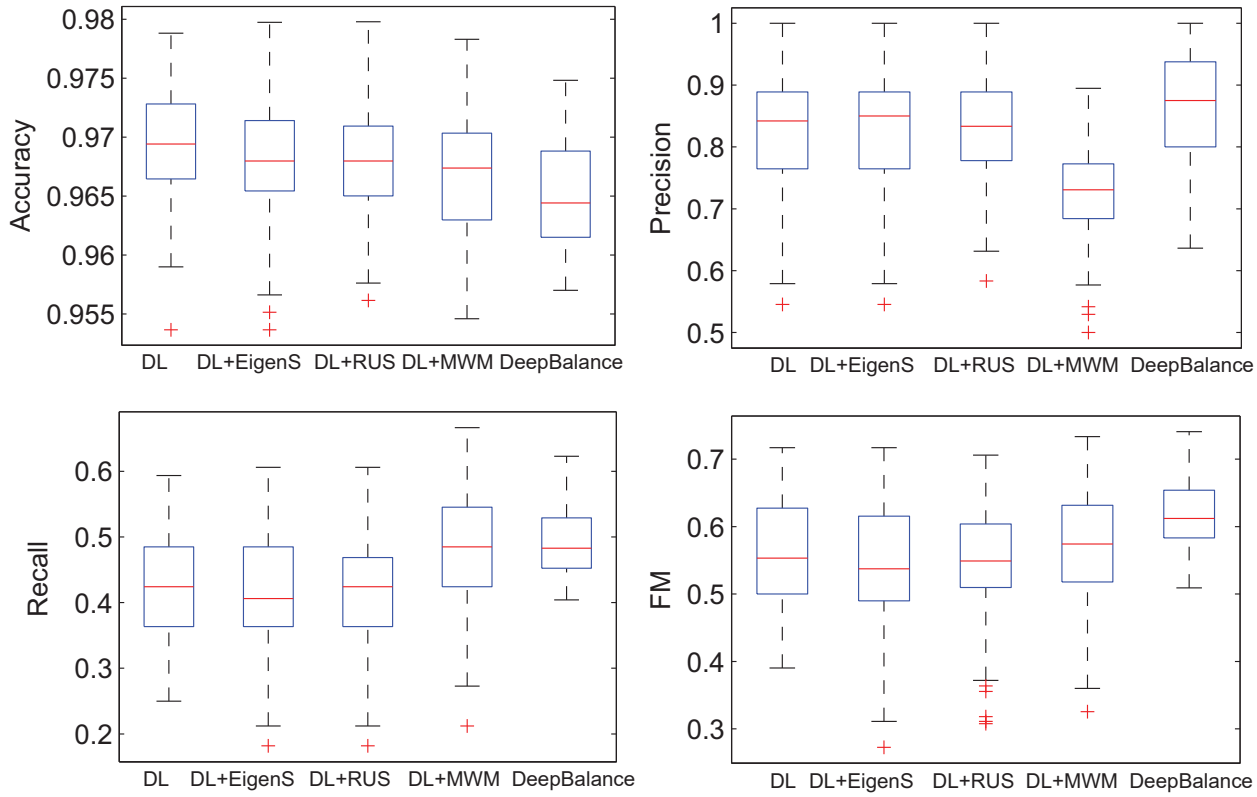


Fig. 8: The Software vulnerability detection results using IR = 20.

Figure Notes: In terms of Precision, DL+MWMOTE (with about 74%) even performs worse than DL (85% on average). However, DeepBalance can improve the performance to 88%. The Recall value of DeepBalance is very comparative to MWMOTE. The FM value of DeepBalance is the best among all the techniques (e.g., the FM value of DeepBalance is at least 6 percent higher than DL).

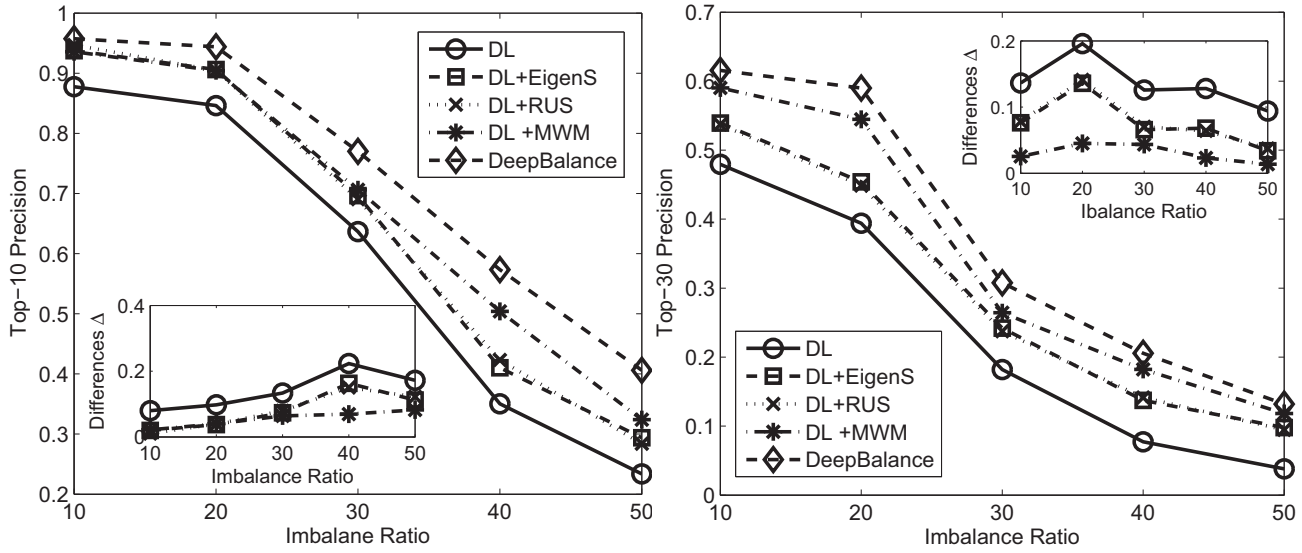
DL+RUS and DL+EigenS drop from about 84% (IR=10) TO about 75% (IR=50). DL+MWM even performs the worst with Precision values at least 8% lower than DL.

Fig.7a depicts the averaged Recall values of the five approaches in terms of IR ranges from 10 to 50. One can observe that DL exhibits the worst performance in all cases. The Recall values achieved by DL is at least 5% less than DeepBalance and DL+MWMOTE. Even though the averaged Recall values of DL+MWMOTE is about 4% higher than DeepBalance when IR = 30, DeepBalance results in very comparative Recall values for other cases. For example, when IR = 20, the Recall value of DeepBalance is about 48.5% which is the same as DL+MWMOTE. Fig.6b indicates that DeepBalance and DL+MWMOTE can identify similarly amount of vulnerabilities when IR between 10 and 50. However, Fig.6b and Fig.7a indicate that DL+MWMOTE achieves better Recall performance at the cost of Precision, which means DL+MWMOTE causes higher false positives.

Fig.7b presents the averaged FM values of the three approaches in terms of IR ranges from 10 to 50. One can see that DeepBalance outperforms all other approaches (except for DL+MWMOTE at IR=30). For example, DeepBalance yields an average FM value of 63.5% when IR = 20, while the classifiers trained from the imbalanced data directly obtain the worst FM value at about 55%. The data processed by DL+MWMOTE method raises the FM value to only 57% which is 6.5% lower than DeepBalance. Although DL+MWMOTE achieves very comparative FM values when IR = 30, its Precision value is about 20% less than DeepBalance. It would be fair to say that DL+MWMOTE fails to build a robust classification model.

Interestingly, we can see that the simple deep learning model has higher accuracy compared to other approaches, while having lower precision and recall. Low precision reflects that the model has high false positive values (recognize non-vulnerable functions as vulnerable), and low recall indicates that simple deep learning model has high false negative values (recognize vulnerable functions as non-vulnerable). The reasons are two-fold: 1) The class imbalance problem leads simple deep learning has higher accuracy with the increase of imbalance ratio. For example, with the increase of imbalance ratio, the classifier normally biased towards the majority class (i.e., the non-vulnerable class). Therefore, the accuracy increased simply because the simple deep learning recognizes most of the functions as non-vulnerable; 2) The total amount of vulnerable functions is small. In this case, a small number of false positives and false negatives will cause low precision and recall. For example, there are only 130 vulnerable functions compared with 6534 non-vulnerable functions when IB = 50. In the one of the test process, the number of TP is 3, TN is 650, FP is 2 and FN is 10. The precision, recall, and FM are only 0.6, 0.231 and 0.333, while the accuracy is as high as

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/TFUZZ.2019.2958558, IEEE Transactions on Fuzzy Systems



(a) IR:10-50 (b) IR:10-50

Fig. 9: The Top-10 and Top-30 Precision values varying IR. DeepBalance outperforms the DL by 15% on average in terms of top-10 Precision. The FM value of DeepBalance is about 6% higher than the second best technique, DL+MWMOTE. The top-30 Precision values of DeepBalance is about 15% higher than DL.

0.982.

Next, we present the experimental results at the specific situation where neural data samples are 20 times of vulnerable data samples (i.e., IR = 10). Fig.8 details the vulnerable detection results based on the data set of IR = 20. It can be seen that DeepBalance obtains very comparative Accuracy values, which is only 0.005 less than the best technique DL in general. In terms of Precision, directly learning from data sets with an imbalanced rate of 20 (DL) yields 84% on average. DL+MWMOTE even performs worse than DL (i.e., the Precision value of DL+MWMOTE is about 72%, which is about 12% less than DL). However, DeepBalance can improve the performance to 88%. Moreover, the medium Recall value of DeepBalance and DL+MWMTOE are at around 50%, which is much more better than DL at about 43%. The last subgraph of Fig.8 is FM, we can see that DeepBalance outperforms other samples techniques, while DL performs the worst. For example, DeepBalance produces an average FM of 61.5% which is slightly higher than the second best technique, DL+MWMOTE at 58%. DL+Eigensample obtaines the worst FM value at around 53%.

Fig.9a depicts the top-10 Precision values. We can see when retrieving 10 most probable vulnerable functions, DeepBalance outperforms other sampling techniques and DL for all cases. Precisely, with the increase of IR, the top-10 precision of DL and other techniques decrease dramatically from 89% and 94% (when IR = 10) to less than 35% (when IR = 50). However, DeepBalance shows much better performance. Take IR = 40 for example, DeepBalance can successfully identify 6 vulnerabilities out of 10, while EigenSample and DL can only identify 4 and 3 vulnerabilities. When have a look at the top-30 results from Fig.9b, one can see DeepBalance also performs the best among other methods. For example, when IR = 20, DeepBalance can identify 61 percent vulnerable functions, while DL can only identify about 40% vulnerabilities.

All in all, the results show that the classifier trained on original imbalance data for software vulnerable detection performs the worst. It classifies more neural functions as vulnerable functions and meanwhile identifies more vulnerable functions as neural functions. While the proposed DeepBalance system yields better performance compared with DL and other samples techniques. Even though DL+MWMOTE obtains very comparative Recall values, it achieves high Recall values at cost false positives. This means that the comparative Recall values obtained by DL+MWMOTE sacrifice the Precision measure, which leads to more false positives.

2) Experiments for answering Q2: To answer Q2, we will compare DeepBalance with three recently developed software vulnerability detection systems, which are G-VulD [47], VulDeePecker [48], and LSTM-FL [33]. These three systems are chosen because they are deep learning based detection approach and are most related to our work. In addition, in order to check whether our proposed system outperforms auto-encoders and rough auto-encoders learning models, stacked autoencoders [56] and rough auto-encoders [56] have been considered in this paper as well. For rough auto-encoder, outputs of the first layer (i.e., the upper bound and lower bound) can be calculate as:

$$h_U^k = Max(f^k(W_U^k X + b_U^k), f^k(W_L^k X + b_L^k))$$
$$h_L^k = Min(f^k(W_U^k X + b_U^k), f^k(W_L^k X + b_L^k))$$

while the hidden layer outputs can be calculated as:

$$h^k = \alpha^k h_U^k + \beta^k h_L^k$$

where W_U^k, b_U^k, and α^k are the parameters of upper boun and W_L^k, b_L^k, and β^k are the parameters of lower bound.

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/TFUZZ.2019.2958558, IEEE Transactions on Fuzzy Systems

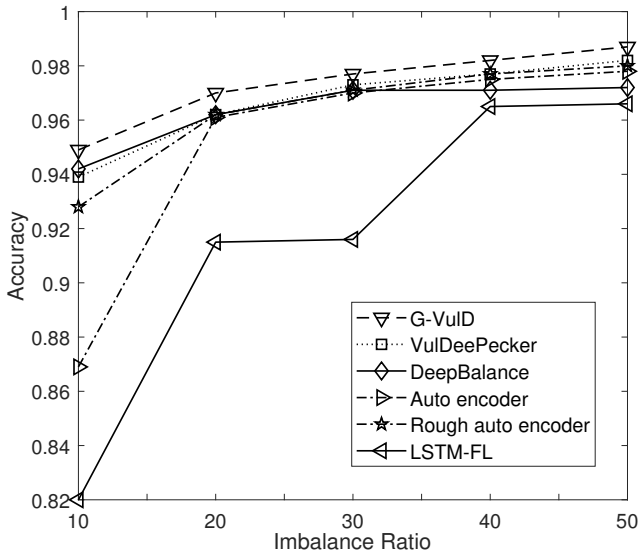


Fig. 10: The Accuracy values varying IR. G-VulD performs the best in terms of Accuracy. Howevever, DeepBalance achieves very comparative Accuracy values.

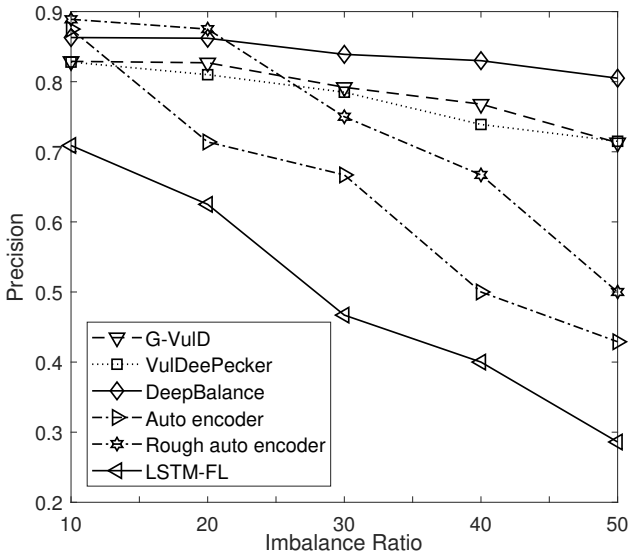


Fig. 11: The Precision values varying IR. DeepBalance can improve the Precision values for at least 10% and 4% compared to VulDeePecker and G-VulD.

Fig.10, Fig.11, Fig.12, and Fig.13 present the Accuracy, Precision, Recall, and FM values of G-VulD, VulDeePecker, DeepBalance, auto-encoders, rough auto-encoders and LSTM-FL approaches in terms of IR rages from 10 to 50. Fig.10 indicates that all the accuracy values of the six approaches increase slightly with the increase of the IR, while Fig.11, Fig.12, and Fig.13 show that the values of the other three metrics suffer a decrease with the increase of the IR. We observe that auto-encoder and rough auto-encoder perform nearly the worst in terms of Recall and FM metrics compared with other approaches. For example, auto-encoders and rough auto-encoders have similar recall values of about 0.15 with FM at around 0.235 and 0.240, which are much lower than DeepBalance (with recall of 0.360 and FM of 0.415). This means auto-encoder and rough auto-encoder cause high false negatives and missed many vulnerable functions. Although rough auto-encoders betters DeepBalance in terms of Precision when IB is 10 and 20, its recalls are about 0.248 and 0.240 when IB is 10 and 20, which is at least 0.25 less than DeepBalance (with 0.548 and 0.490, respectively). This indicates that rough auto-encoders achieves better precision at the cost of recall, which again emphasized that rough auto-encoders missed many vulnerable functions.

Fig.10 also demonstrates that outperforms LSTM-FL in terms of all performance measures. LSTM-FL results in poor performance in recall and FM because of the lowest accuracy and precision values it achieved. This means that LSTM-FL causes highly false positives and misclassified many non-vulnerable functions as vulnerable functions. In addition, we also observe that VulDeePecker performs the worst in terms of Precision, Recall and FM metrics compared with G-VulD and DeepBalance. This can be explained by the fact that when facing the class imbalance problem, the classification results are biased towards the not vulnerable functions, which results in very poor Recall values (True positive rate) and missed a lot of vulnerable functions. Therefore, VulDeePecker is not able to deal with the class imbalance problem in software vulnerability detection.

The proposed DeepBalance outperforms the seq2seq LSTM structure and the autoencoders. We suspect this is because the Bi-LSTM structure is more suitable for learning long-range dependencies which are crucial for identifying vulnerable code snippets. As discussed in Section IV-A, vulnerable code patterns usually consist of multiple code statements, which are either related to preceding or subsequent context or even both. The bidirectional form of the LSTM implementation facilitates the learning of both forward and backward directions of the code sequences, therefore are effective for capturing the vulnerable code patterns being context-dependent. The LSTM-based seq2seq structure, however, can only capture a shorter code context dependency compared with the bidirection structure. In addition, the seq2seq structure provided by Dam et al. [33] enables the code token representations to be learned during the training process of the network, which requires the model to be more expressive to learn both the representations of the code tokens and the model parameters. Compared with our method, the autoencoder was underperformed in our data sets. We conjecture the autoencoder network is constructed with multiple fully-connected dense layers which are designed for learning high-level features of code. It is able to capture the latent patterns of the code semantics and syntactic, but the fully-connected structure is not optimized for learning sequential data that are context-aware and interrelated.

All in all, from Fig.10, Fig.11, Fig.12, and Fig.13, we can see that DeepBalance results in higher Recall, Precision and FM values. Higher Recall values mean better true positive rate. Higher Precision values mean lower false negative rate, and higher FM values mean DeepBalance has a better overall

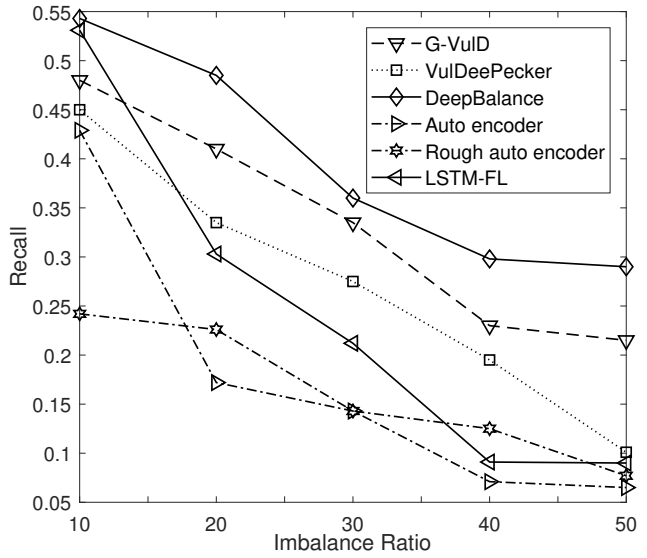


Fig. 12: The Recall values varying IR. DeepBalance improves the Recall values up to 6% and 10% compared with G-VulD and VulDeePecker.

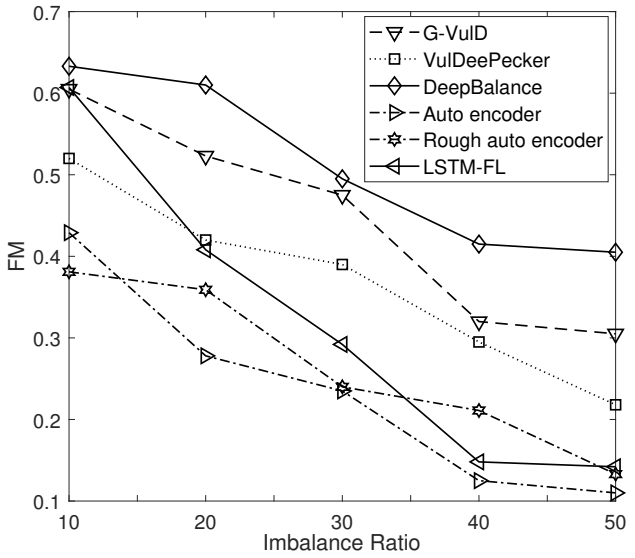


Fig. 13: The FM values varying IR. The FM values of DeepBalance is at least 6% and 12% higher than G-VulD and VulDeePecker.

classification performance. Therefore, we can conclude that DeepBalance can detect more vulnerabilities with lower false negatives and higher true positive rate. This indicates that DeepBalance can deal with the class imbalance problem in software vulnerability detection, which emphasized the effectiveness of DeepBalance approach.

VI. Conclusion

In this paper, we proposed the DeepBalance system for software vulnerability detection based on deep learning and fuzzy oversampling. The goal is to deal with the class imbalance problem when using existing source code with known vulnerabilities to build classification model for detecting vulnerable functions in a new project. To achieve this goal, DeepBalance first employs a deep learning network architecture to learn high-level abstractions on the open-source programs. Then, a fuzzy oversampling algorithm is proposed to deal with the class imbalance problem where the number of non-vulnerable functions accounts for the majority of the total function samples in the training data set. Extensive experiments have been conducted using the collected open source projects data. Experiments show that DeepBalance not only outperforms the simple deep learning method (by at least 5% in terms of FM) but also the combined method (by at least 2% in terms of FM). When compared with other machine learning methods, DeepBalance can still improve FM by 5% in general, especially compared with autoencoders and rough auto-encoders.

As far as we know, DeepBalance is the first deep learning system that presented to deal with the class imbalance problem in software vulnerability detection. There are several directions can be further explored. Firstly, we plan to apply our proposed system to software fault detection. Secondly, we will extend the proposed FOS by taking the inter-correlations among features into account. This is because FOS assumes the data samples are independent of each other in the process of generating synthetic vulnerable data samples, which may not always be true in real-world scenarios. Lastly, applying DeepBalance for detecting venerability at binary-level would be an interesting direction.

Acknowledgement

This work was supported by resources provided by the Pawsey Supercomputing Centre with funding from the Australian Government and the Government of Western Australia.

References

[1] X. Chen, C. Li, D. Wang, S. Wen, J. Zhang, S. Nepal, Y. Xiang, and K. Ren, "Android hiv: A study of repackaging malware for evading machine-learning detection," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 987–1001, 2019.

[2] 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.

[3] N. Sun, J. Zhang, P. Rimba, S. Gao, Y. Xiang, and L. Y. Zhang, "Data-driven cybersecurity incident prediction: A survey," *IEEE Communications Surveys & Tutorials*, 2018.

[4] 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 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 426–437.

[5] S. Zhang, X. Ou, and D. Caragea, "Predicting cyber risks through national vulnerability database," *Information Security Journal: A Global Perspective*, vol. 24, no. 4-6, pp. 194–206, 2015.

[6] H. Alves, B. Fonseca, and N. Antunes, "Experimenting machine learning techniques to predict vulnerabilities," in *Dependable Computing (LADC), 2016 Seventh Latin-American Symposium on*. IEEE, 2016, pp. 151–156.

[7] Y. Younan, "25 years of vulnerabilities: 1988–2012," *Sourcefire Vulnerability Research Team*, 2013.

[8] R. Coulter, Q.-L. Han, L. Pan, J. Zhang, and Y. Xiang, "Data-driven cyber security in perspective–intelligent traffic analysis," *IEEE transactions on cybernetics*, 2019.

[9] K. Soska and N. Christin, "Automatically detecting vulnerable websites before they turn malicious." in *USENIX Security Symposium*, 2014, pp. 625–640.

[10] A. Younis, Y. Malaiya, C. Anderson, and I. Ray, "To fear or not to fear that is the question: Code characteristics of a vulnerable functionwith an existing exploit," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. ACM, 2016, pp. 97–104.

[11] S. Huda, K. Liu, M. Abdelrazek, A. Ibrahim, S. Alyahya, H. Al-Dossari, and S. Ahmad, "An ensemble oversampling model for class imbalance problem in software defect prediction," *IEEE access*, vol. 6, pp. 24 184–24 195, 2018.

[12] 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.

[13] C. Sabottke, O. Suciu, and T. Dumitras, "Vulnerability disclosure in the age of social media: Exploiting twitter for predicting real-world exploits." in *USENIX Security Symposium*, 2015, pp. 1041–1056.

[14] P. Morrison, K. Herzig, B. Murphy, and L. Williams, "Challenges with applying vulnerability prediction models," in *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*. ACM, 2015, p. 4.

[15] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey," *ACM Computing Surveys (CSUR)*, vol. 50, no. 4, p. 56, 2017.

[16] S. Liu, J. Zhang, Y. Xiang, and W. Zhou, "Fuzzy-based information decomposition for incomplete and imbalanced data learning," *IEEE Transactions on Fuzzy Systems*, vol. 25, no. 6, pp. 1476–1490, 2017.

[17] T. Wu, S. Wen, Y. Xiang, and W. Zhou, "Twitter spam detection: Survey of new approaches and comparative study," *Computers & Security*, vol. 76, pp. 265–284, 2018.

[18] S. Liu, M. Dibaei, Y. Tai, C. Chen, J. Zhang, and Y. Xiang, "Cyber vulnerability intelligence for iot binary," *IEEE Transactions on Industrial Informatics*, 2019.

[19] G. Lin, J. Zhang, W. Luo, L. Pan, O. De Vel, P. Montague, and Y. Xiang, "Software vulnerability discovery via learning multi-domain knowledge bases," *IEEE Transactions on Dependable and Secure Computing*, 2019, in press, DOI:http://doi.org/10.1109/TDSC.2019.2954088.

[20] L. K. Shar and H. B. K. Tan, "Predicting sql injection and cross site scripting vulnerabilities through mining input sanitization patterns," *Information and Software Technology*, vol. 55, no. 10, pp. 1767–1780, 2013.

[21] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 797–812.

[22] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient cross-architecture identification of bugs in binary code." in *NDSS*, 2016.

[23] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. ACM, 2016, pp. 85–96.

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

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

[26] 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, pp. 13–13.

[27] W. N. Poon, K. E. Bennin, J. Huang, P. Phannachitta, and J. W. Keung, "Cross-project defect prediction using a credibility theory based naive bayes classifier," in *Software Quality, Reliability and Security (QRS), 2017 IEEE International Conference on*. IEEE, 2017, pp. 434–441.

[28] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 382–391.

[29] S. J. Pan, I. W. Tsang, J. T. Kwok, and Q. Yang, "Domain adaptation via transfer component analysis," *IEEE Transactions on Neural Networks*, vol. 22, no. 2, pp. 199–210, 2011.

[30] 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.

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

[32] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, 2014.

[33] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for vulnerability prediction," *arXiv preprint arXiv:1708.02368*, 2017.

[34] M. Pinzger, N. Nagappan, and B. Murphy, "Can developer-module networks predict failures?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008, pp. 2–12.

[35] A. Meneely and L. Williams, "Secure open source collaboration: an empirical study of linus' law," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 453–462.

[36] S. Liu, Y. Wang, J. Zhang, C. Chen, and Y. Xiang, "Addressing the class imbalance problem in twitter spam detection using ensemble learning," *Computers & Security*, vol. 69, pp. 35–49, 2017.

[37] S. Chen, H. He, and E. A. Garcia, "Ramoboost: ranked minority oversampling in boosting," *IEEE Transactions on Neural Networks*, vol. 21, no. 10, pp. 1624–1642, 2010.

[38] M. Bekkar and T. A. Alitouche, "Imbalanced data learning approaches review," *International Journal of Data Mining & Knowledge Management Process*, vol. 3, no. 4, p. 15, 2013.

[39] H. He, Y. Bai, E. A. Garcia, and S. Li, "Adasyn: Adaptive synthetic sampling approach for imbalanced learning," in *IEEE International Joint Conference on Neural Networks*. IEEE, 2008, pp. 1322–1328.

[40] T. Jo and N. Japkowicz, "Class imbalances versus small disjuncts," *ACM Sigkdd Explorations Newsletter*, vol. 6, no. 1, pp. 40–49, 2004.

[41] S. Barua, M. M. Islam, X. Yao, and K. Murase, "Mwmote–majority weighted minority oversampling technique for imbalanced data set learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 2, pp. 405–425, 2014.

[42] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[43] M. Hao, Y. Wang, and S. H. Bryant, "An efficient algorithm coupled with synthetic minority over-sampling technique to classify imbalanced pubchem bioassay data," *Analytica chimica acta*, vol. 806, pp. 117–127, 2014.

[44] H. Zhao, X. Chen, T. Nguyen, J. Z. Huang, G. Williams, and H. Chen, "Stratified over-sampling bagging method for random forests on imbalanced data," in *Pacific-Asia Workshop on Intelligence and Security Informatics*. Springer, 2016, pp. 63–72.

[45] R. C. Prati, G. E. Batista, and D. F. Silva, "Class imbalance revisited: a new experimental setup to assess the performance of treatment methods," *Knowledge and Information Systems*, vol. 45, no. 1, pp. 247–270, 2015.

[46] X. Ban, S. Liu, C. Chen, and C. Chua, "A performance evaluation of deep-learnt features for software vulnerability detection," *Concurrency and Computation: Practice and Experience*, p. e5103, 2019.

[47] 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 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2017, pp. 2539–2541.

[48] 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," *arXiv preprint arXiv:1801.01681*, 2018.

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

[50] L. Moonen, "Generating robust parsers using island grammars," in *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*. IEEE, 2001, pp. 13–22.

[51] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *1st International Conference on Learning Representations, Workshop Track Proceedings*, 2013.

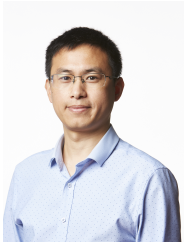
[52] G. Casella and R. L. Berger, *Statistical inference*. Duxbury Pacific Grove, CA, 2002, vol. 2.

[53] 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*, vol. 14, no. 7, pp. 3289–3297, 2018.

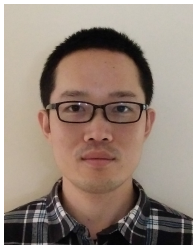
[54] J. Zhang, X. Chen, Y. Xiang, W. Zhou, and J. Wu, "Robust network traffic classification," *IEEE/ACM Transactions on Networking (TON)*, vol. 23, no. 4, pp. 1257–1270, 2015.

[55] S. Soman, S. Saxena *et al.*, "Eigensample: A non-iterative technique for adding samples to small datasets," *Applied Soft Computing*, 2017.

[56] M. Khodayar, O. Kaynak, and M. E. Khodayar, "Rough deep neural architecture for short-term wind speed forecasting," *IEEE Transactions on Industrial Informatics*, vol. 13, no. 6, pp. 2770–2779, 2017.



Shigang Liu (M'15) received his PhD in Computer Science from Deakin University, Australia, in 2017. Currently, he is a second-year research fellow with School of Software and Electrical Engineering, Swinburne University of Technology. His research mainly focuses on security, applied machine learning, and fuzzy information processing. He is currently leading a research group on applying deep learning to detect software vulnerabilities. This research topic won the first place of the World Change Maker Prize in 2019 Swinburne Research Conference. He has published more than 20 research papers in many international journals and conferences. He is a member of the Editorial Board of the Journal of Mathematics and Informatics (JMI). He has been the Publication Chair, Local Arrangement Chair and PC member for several conferences including CSS2017, CSDE2019, ML4CS2019 and so on.



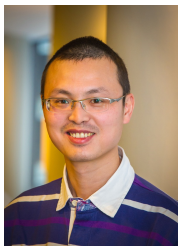
Guanjun Lin Guanjun Lin received the bachelor's degree in information technology (with first class Hons.) from Deakin University, Geelong, VIC., Australia, in 2012. He is currently working toward the Ph.D degree in the School of Software and Electrical Engineering at the Swinburne university of technology.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

Qing-Long Han (M'09-SM'13-F'19) received the B.Sc. degree in Mathematics from Shandong Normal University, Jinan, China, in 1983, and the M.Sc. and Ph.D. degrees in Control Engineering and Electrical Engineering from East China University of Science and Technology, Shanghai, China, in 1992 and 1997, respectively.

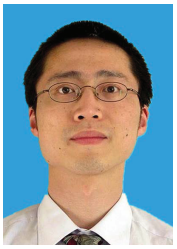
From September 1997 to December 1998, he was a Post-doctoral Researcher Fellow with the Laboratoire d'Automatique et d'Informatique Industielle (currently, Laboratoire d'Informatique et d'Automatique pour les Systémes), École Supérieure d'Ingénieurs de Poitiers (currently, École Nationale Supérieure d'Ingénieurs de Poitiers), Université de Poitiers, France. From January 1999 to August 2001, he was a Research Assistant Professor with the Department of Mechanical and Industrial Engineering at Southern Illinois University at Edwardsville, USA. From September 2001 to December 2014, he was Laureate Professor, an Associate Dean (Research and Innovation) with the Higher Education Division, and the Founding Director of the Centre for Intelligent and Networked Systems at Central Queensland University, Australia. From December 2014 to May 2016, he was Deputy Dean (Research), with the Griffith Sciences, and a Professor with the Griffith School of Engineering, Griffith University, Australia. In May 2016, he joined Swinburne University of Technology, Australia, where he is currently Pro Vice-Chancellor (Research Quality) and a Distinguished Professor. His research interests include networked control systems, multi-agent systems, time-delay systems, complex dynamical systems and neural networks.

Professor Han is a Highly Cited Researcher according to Clarivate Analytics (formerly Thomson Reuters). He is a Fellow of The Institution of Engineers Australia. He is an Associate Editor of several international journals, including the IEEE TRANSACTIONS ON CYBERNETICS, the IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS, the IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS, IEEE INDUSTRIAL ELECTRONICS MAGAZINE, the IEEE/CAA JOURNAL OF AUTOMATICA SINICA, Control Engineering Practice, and Information Sciences.

Sheng Wen (M'15) received his PhD degree from Deakin University, Melbourne, in October 2014. Currently he has been working fill-time as a senior lecturer in Swinburne University of Technology. Before this, he first worked as a research fellow and then a Lecturer in Computer Science in the School of Information Technology in Deakin University from the year of 2015. Dr. Wen manages several research projects in the last three years. Since late 2014, Dr. Wen has received over 3 million Australia Dollars funding from both academia and industries. Dr. Wen is also leading a medium-size research team (around 15 members) in cybersecurity area. He has been invited to be Chair Committee members for CSS 2017, SocialSec 2017, MONAMI 2017, WMNC 2015, CSS 2015, IEEE BigDataService 2015, and GPC 2015. He also served as PC member for a number of International Conferences, such as Trustcom 2014, NSS 2014, SmartComp 2014, Trustcom 2015, ACISP 2016, AICCSA 2015, CSS 2012, DependSys 2015, HPCC 2015, ICA3PP 2015, ICA3PP 2011, ISICA 2015, NSS 2015, SNAMS 2015, SocialSec 2015, SocialSec 2016, DependSys 2016, TrustCom 2016, IEEE ICC 2016, IEEE Globalcom 2016, etc.. His research interests include IP Network Resilience and Security, Software Security



Jun Zhang (M'12-SM'18) received the Ph.D. degree from the University of Wollongong, Australia, in 2011. He is the Co-founder and Deputy Director of the Cybersecurity Lab, Swinburne University of Technology, Australia. His research interests include cybersecurity and applied machine learning. He is the Chief Investigator of several projects in cybersecurity, funded by the Australian Research Council (ARC). He has published more than 100 research papers in many international journals and conferences. Two of his papers were selected as the featured articles in the July/August 2014 issue of IEEE Transactions on Dependable and Secure Computing and the March/April 2016 issue of IEEE IT Professional. His research has been widely cited in the area of cybersecurity. He has been internationally recognised as an active researcher in cybersecurity, evidenced by his chairing of 15 international conferences, and presenting of invited keynote addresses in 6 conferences and an invited lecture in IEEE SMC Victorian Chapter.



Yang Xiang (M'07-SM'12-F'20) received his Ph.D. in Computer Science from Deakin University, Australia. He is 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. He has published more than 200 research papers in many international journals and conferences, such as IEEE Transactions on Computers, IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Information Security and Forensics, and IEEE Journal on Selected Areas in Communications. He has published two books, Software Similarity and Classification and Dynamic and Advanced Data Mining for Progressing Technological Development. He has served as the Program/General Chair for many international conferences such as SocialSec 15, IEEE DASC 15/14, IEEE UbiSafe 15/14, IEEE TrustCom 13, etc. He has been the PC member for more than 80 international conferences in distributed systems, networking, and security. He served as the Associate Editor of IEEE Transactions on Computers, IEEE Transactions on Parallel and Distributed Systems, Security and Communication Networks (Wiley), and the Editor of Journal of Network and Computer Applications. He is the Coordinator, Asia for IEEE Computer Society Technical Committee on Distributed Processing (TCDP).