CD-VulD: Cross-Domain Vulnerability Discovery Based on Deep Domain Adaptation

Shigang Liu, *Member, IEEE*, Guanjun Lin, Lizhen Qu, Jun Zhang, *Senior Member, IEEE*, Olivier De Vel, Paul Montague, and Yang Xiang, *Fellow, IEEE*

Abstract—A major cause of security incidents such as cyber attacks is rooted in software vulnerabilities. These vulnerabilities should ideally be found and fixed before the code gets deployed. Machine learning-based approaches achieve state-of-the-art performance in capturing vulnerabilities. These methods are predominantly supervised. Their prediction models are trained on a set of ground truth data where the training data and test data are assumed to be drawn from the same probability distribution. However, in practice, the test data often differs from the training data in terms of distribution because they are from different projects or they differ in the types of vulnerability. In this article, we present a new system for Cross Domain Software Vulnerability Discovery (*CD-VulD*) using deep learning (DL) and domain adaptation (DA). We employ DL because it has the capacity of automatically constructing high-level abstract feature representations of programs, which are likely of more cross-domain useful than the handcrafted features driven by domain knowledge. The divergence between distributions is reduced by learning cross-domain representations. First, given software program representations, CD-VulD converts them into token sequences and learns the token embeddings for generalization across tokens. Next, CD-VulD employs a deep feature model to build abstract high-level presentations based on those sequences. Then, the metric transfer learning framework (MTLF) technique is employed to learn cross-domain representations by minimizing the distribution divergence between the source domain and the target domain. Finally, the cross-domain representations are used to build a classifier for vulnerability detection. Experimental results show that CD-VulD outperforms the state-of-the-art vulnerability detection approaches by a wide margin. We make the new datasets publicly available so that our work is replicable and can be further improved.

Index Terms—Cross-domain, vulnerability detection/discovery, deep learning, machine learning, domain adaptation

1 Introduction

Software vulnerability detection (SVD) is a challenging problem for mitigating security risks in software [1]. Recent studies found that the fast growth of digital services and products in the market leads to a dramatic increase in software vulnerabilities [2], [3]. For example, the well-known Common Vulnerabilities and Exposures (CVE) database only registered around 4,500 vulnerabilities in 2010 and the number increased to over 8,000 in 2014 [4]. In 2017, this has risen to about 15,000 vulnerabilities in the CVE database. These vulnerabilities affected the secure usage of more than 17,000 digital services/products (e.g., end-user software, and device firmware), and caused direct financial losses of 226 billion dollars a year [5]. These vulnerabilities should ideally have been found and removed by security audits [6], [7], [8].

- *S. Liu, G. Lin, J. Zhang, and Y. Xiang are with the School of Software and Electrical Engineering, Swinburne University of Technology, Hawthorn, VIC 3122, Australia. E-mail: {shigangliu, glin, junzhang, yxiang}@swin.edu.au.*
- *L. Qu is with Data61, Docklands, VIC 3008, Australia. E-mail: lizhen.qu@data61.csiro.au.*
- *O. De Vel and P. Montague are with the Defence Science & Technology Group, Edinburgh, SA 5111, Australia. E-mail: {olivier.devel, paul.montague}@dst.defence.gov.au.*

Manuscript received 24 Sept. 2018; revised 15 Jan. 2020; accepted 7 Mar. 2020. Date of publication 2 Apr. 2020; date of current version 17 Jan. 2022. (Corresponding author: Jun Zhang.)
Digital Object Identifier no. 10.1109/TDSC.2020.2984505

There are ample techniques proposed for software vulnerability detection. They largely fall into two categories: learning based and non-learning based techniques. The non-learning based techniques can be further categorized into static, dynamic and hybrid approaches. The static techniques such as code similarity detection (i.e., code clone detection) [2], [9], symbolic execution [10] and rule-based analysis [11] mainly focus on the analysis of source code, and are usually not able to reveal bugs and vulnerabilities occurring at run time. Dynamic analysis (e.g., fuzzing [12] and taint analysis [13]) identifies vulnerabilities during the execution of a program, but in general has low code coverage. Hybrid approaches which combine static and dynamic analysis techniques for vulnerability detection are expected to overcome the aforementioned weaknesses. However, hybrid approaches heavily rely on a limited set of known syntactic or behavioral patterns of vulnerabilities which requires longer computation times and larger resource requirements [14]. The machine learning-based systems are able to achieve superior performance than non-learning based methods [2], [3], [4], [15], [16]. Those learning-based systems predominantly apply supervised learning techniques [17], [18], [19], which train predictive models on a set of ground truth datasets. The training and test datasets are assumed to be drawn from the same distribution.

In practice, the distribution assumption is often violated because there are both emerging novel vulnerability types and new projects containing known vulnerability types. In

both cases, the models trained on known vulnerability types and existing projects often fail to predict new vulnerabilities because they do not follow the same distributions as the models' training data. This phenomenon is often referred to as the Cross-Domain problem by considering that training data come from a source domain while the test data is from a target domain. The two domains have different distributions because the corresponding data contain vulnerabilities of different types or from different projects. Our independent studies show that simply applying the top performing models such as G-VulD [16] and VulDeePecker [15] trained on source domain data incur more than 80 percent false positive rate (FPR) and false negative rate (FNR) on vulnerabilities in a target domain. The alternative approach is to build new training data based on vulnerabilities of new types or from new projects. However, the construction of such datasets is too expensive and time-consuming.

In light of the above analysis, we developed the *C*ross-*D*omain *V*ulnerability *D*etection (CD-VulD) system, which is the first work that addresses the cross-domain issue in vulnerability detection. This system assumes that there is a handful of labeled data in the target domain and there is also a large amount of labeled training data in the source domain. Specifically, the source and target domain can be either a data set with single type of vulnerability (e.g., CWE119 of buffer errors) or a project with multiple of vulnerabilities (e.g., LibTIFF of numeric errors, null-pointer dereference, format string vulnerability etc.). We first convert functions in source code into token sequences, which are then mapped to high-level representations by applying deep learning techniques. In order to reduce the performance loss caused by distribution divergence, we extend the metric transfer learning framework (MTLF) [20] to learn cross-domain representations of token sequences, so that we only need to train a classifier on the source domain labeled data and apply the trained classifier to the target domain. We demonstrate the effectiveness of this system by extensive experiments in the settings of cross vulnerability types or cross projects. Our main contributions are summarized as follows:

First, our work is the first one that studies vulnerability detection in the cross-domain setting. We provide empirical evidence to show that prior deep learning models on this task suffer from more than 80 percent error rates.

Second, our system CD-VulD significantly outperforms the existing systems in the following cross-domain settings:

- The source domain data and the target domain data are from the same project, but the corresponding vulnerabilities are of different types.
- The source domain data and the target domain data are from different projects. Each domain contains only a single vulnerability type. This setting is very helpful for some vulnerability types with few examples.
- The source domain data and the target domain data are from different projects. Each domain contains multiple vulnerability types. This is the most common scenario when we try to leverage existing training data for vulnerability detection in new projects.
- The source domain data and the target domain data are from different projects with different features. This is a study of invariance with respect to syntactic representations in the cross-domain setting.

Third, we experimentally demonstrate the robustness of the proposed CD-VulD by applying it in the in-domain setting. It achieves comparable results to the existing machine learning based solutions.

Finally, we make available the statistical datasets for other researchers to use and contribute to the cross-domain SVD problem. The datasets which are learned from DL, contain the data from both [16] and [15]. The datasets are available at https://github.com/wolong3385/SVD-Source.

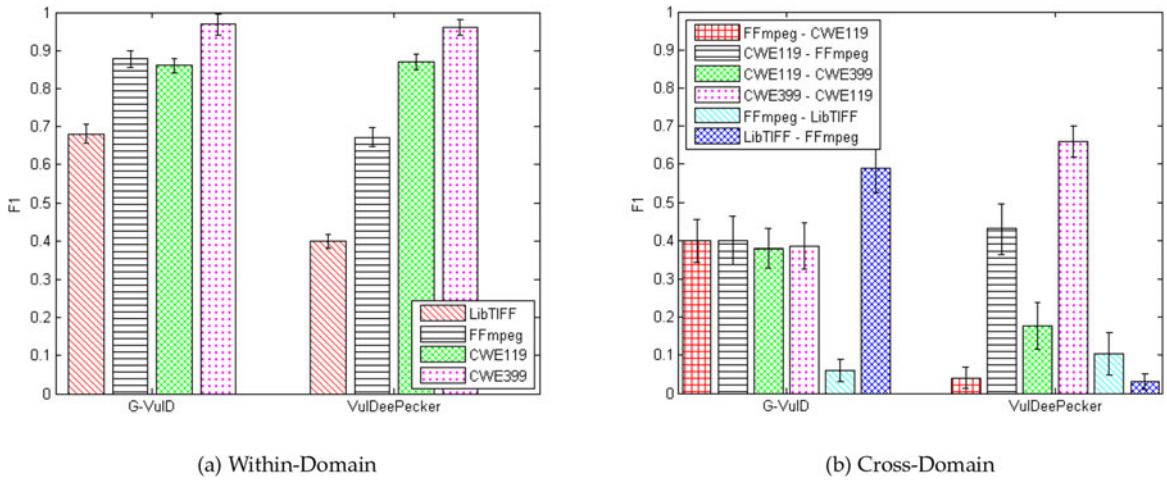
The rest of the paper is organized as follows. Section 2 shows the problems of prior deep learning models in the cross-domain setting. Section 3 presents our proposed system. The experimental study and results are described in Section 4, and their limitations discussed in Section 5. Section 6 reviews the related work, and finally Section 7 concludes this paper.

2 Difficulties of Cross-Domain Vulnerability Detection

In this section, we demonstrate the difficulties of cross-domain vulnerability detection with two state-of-the-art deep learning systems: G-VulD and VulDeePecker. We evaluate both systems on four datasets: CWE119, CWE399, FFmpeg, and LibTIFF. Both CWE119 and CWE399 contain a single type of vulnerability, which is buffer error and resource management error respectively. Both FFmpeg and LibTIFF have the same mixed types of vulnerabilities, including buffer overflow and resource management errors. We take the F1-score (F1) as the evaluation metric because it reflects both FPR and FNR. As a reference, we evaluate how well each system performs in the in-domain setting. We first split each dataset into a training and a test set with ratio 70 to 30 percent. Then we train G-VulD and VulDeePecker on the training set and evaluate the models on the test set for each dataset. As shown in Fig. 1a, both systems are able to achieve reasonable F1-scores. LibTIFF appears to be the most difficult dataset, while CWE399 is the easiest one.

In the cross-domain setting, we create multiple pairs of datasets by taking one of them as the training set with the other as the test set. We use the notation XXX-YYY to indicate training on XXX and testing on YYY. Note that, in the cases CWE119-CWE399 and CWE399-CWE119, there is a single type per project and they are of different types. FFmpeg-LibTIFF and LibTIFF-FFmpeg consist of projects of mixed types. In addition, the training and test sets of FFmpeg-CWE119 and CWE119-FFmpeg differ in program representations. FFmpeg uses ASTs, while CWE119 uses code gadget.

From Fig. 1b we can see that both systems in the cross-domain setting suffer from much lower F-Measures than in the in-domain setting. In the easiest case, where each project contains one vulnerability type, there is a drop of performance for both systems. When projects contain mixed types of vulnerability, it becomes too challenging for VulDeePecker and the F-Measures are lower than 0.1. As a comparison, G-VulD can work much better than VulDeePecker when FFmpeg is



(a) Within-Domain (b) Cross-Domain

Fig. 1. Classification performance: Within-Domain versus Cross-Domain.

used as the test set, but it is still much worse than the in-domain setting. When program representations are changed, VulDeePecker can hardly work in one of the scenarios. Overall, both deep learning based systems cannot achieve satisfactory results in the cross-domain setting.

3 The Proposed Methodology: CD-VulD

The key idea of our proposed system CD-VulD is to learn cross-domain representations of source code at the function level. A domain refers to either a project or a set of vulnerability types. We assume that there is a large amount of labeled data in the source domain, thus we are able to train a classifier on the cross-domain representations of functions and apply it to the functions in the target domain. The classifier is able to work in the target domain because the cross-domain representations aim to minimize the discrepancy of distributions between the source domain and the target domain.

Formally, a domain $\mathcal{D} = \{\mathcal{X}, \mathcal{Y}, P(\mathcal{X}, \mathcal{Y})\}$ is characterized by a feature space $\mathcal{X}$, a label space $\mathcal{Y}$, and a joint probability distribution $P(\mathcal{X}, \mathcal{Y})$. The feature space consists of functions in source code. Some of the functions are vulnerable. Their types of vulnerability are from the set $\mathcal{T}$. The label space $\mathcal{Y} = \{1, 0\}$ with 1 indicating vulnerable code and 0 not. Furthermore, let $P_s(\mathcal{X}, \mathcal{Y})$ and $P_t(\mathcal{X}, \mathcal{Y})$ denote the joint distribution in the source domain and the target domain respectively, we have $P_s(\mathcal{X}, \mathcal{Y}) \neq P_t(\mathcal{X}, \mathcal{Y})$. Since $P(\mathcal{X}, \mathcal{Y}) = P(\mathcal{Y}|\mathcal{X})P(\mathcal{X})$, in this work, we further assume that $P_s(\mathcal{Y}|\mathcal{X}) == P_t(\mathcal{Y}|\mathcal{X})$ and $P_s(\mathcal{X}) \neq P_t(\mathcal{X})$. The goal of this work is to learn a vulnerability detection function $f : \mathcal{X} \rightarrow \mathcal{Y}$ by minimizing the discrepancy between $P_s(\mathcal{X})$ and $P_t(\mathcal{X})$.

Due to the assumption $P_s(\mathcal{Y}|\mathcal{X}) == P_t(\mathcal{Y}|\mathcal{X})$, we just need to train a classifier in one domain and use it across domains. The challenge of this domain adaptation problem is to construct representations of $x \in \mathcal{X}$, which encode vulnerability patterns consistently across domains. To circumvent the problem, as shown in Fig. 2, we learn the vulnerability detection function in four stages. First, we preprocess functions into token sequences. Second, we pretrain a deep feature model to map token sequences into

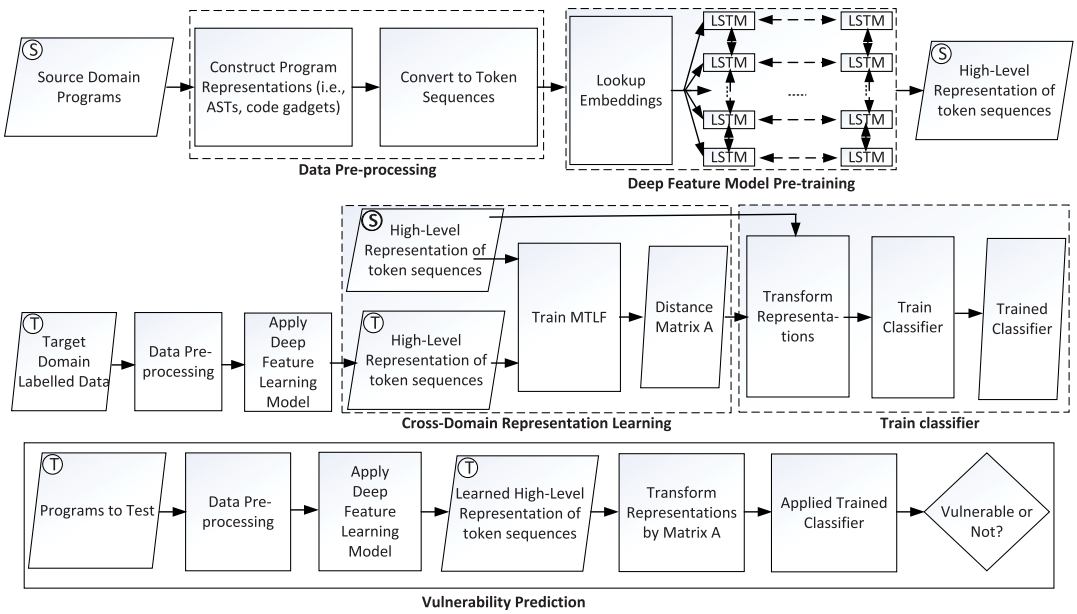
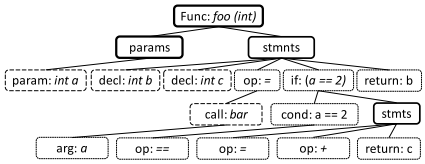


Fig. 2. The framework of CD-VulD.

```
1  int foo (int a)
2  {
3     int c = 0;
4     /* Invoke function bar() */
5     int b = bar (a);
6     if (a == 2)
7     {
8        c = a + b;
9        return c;
10    }
11    return b;
12 }
```

(a) An Example of a C Function.



(b) An AST of function 'foo'.

# 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 == 2)	
cond	3	a == 2	
op	4	==	
stmts	3		
op	4	=	
op	4	+	
return	4	c	
return	2	b	

(c) A serialized AST of function 'foo'.

Fig. 3. Motivating examples of the source code of the function 'foo', its AST and the AST in the serialized format.

dense vectors as their representations. Third, we employ MTLF to learn a transformation matrix to project those dense vectors into a space of cross-domain representations, which reduce the discrepancy between $P_s(\mathcal{X})$ and $P_t(\mathcal{X})$. Last, we learn a classifier on the cross-domain representations by using the source domain data. The details of each stage will be given in the following sections.

After training, for a given function, the CD-VulD first applies pre-processing techniques to turn it into symbolic token sequences, then maps the token sequence into its cross-domain representation by using the trained deep feature model and the transformation matrix. The classifier trained in the source domain data is applied to that representation to predict if it is vulnerable.

3.1 Preprocessing

We consider Abstract Syntax Trees and Code Gadgets as the syntactic representations of software programs.

3.1.1 Abstract Syntax Trees (ASTs)

Programs' ASTs are syntactical structures of source code, which are used to depict the hierarchy of code components and the control flow (i.e., at function level) [21]. In this paper, we use the "CodeSensor", which is a robust parser implemented by [21] to obtain ASTs from source code functions. The "CodeSensor" enables the ASTs extraction to be performed without a working build environment. We follow the method provided by Lin *et al.* [16] to generate the ASTs in a serialized format. The ASTs in the serialized format present a more straightforward view than the original tree view and can be converted to sequences for subsequent processing. Fig. 3 shows the source code of the function *foo*, its AST and the serialized format of the AST.

3.1.2 Code Gadget

We briefly recall the definition of a code gadget, which is a method to represent a slice of source programs.

(Code Gadget) [15] A code gadget is composed of a number of program statements (e.g., lines of code), which are semantically related to each other in terms of data dependency or control dependency.

Code gadgets refer to the key points of the programs that directly cause the vulnerabilities. The key points can be library/API function calls if the vulnerability is caused directly by improper use of library/API function calls; it can be an array if the vulnerability is caused by improper use of arrays; and it can be pointers if the vulnerability is caused by improper use of pointers. For more information and exmaples of code gadget, please refer to [15]. Note that, different kinds of vulnerabilities may share the same key points. For example, both resource management errors and buffer errors may be caused by improper use of library/API function calls. Since the source code of VulDeePecker is not open, and it used commercial products such as Checkmarx[1] for code gadgets' generation, in this paper we simply reuse the code gadgets reported in [15].

3.1.3 Token Sequences

We choose ASTs as an example to discuss the token sequences and we apply depth-first traversal (DFT) for traversing the ASTs as presented in [16]. The resulting ASTs are in serialized format shown in Fig. 3c. Through DFT, the nodes of an AST are mapped to a vector so that each node becomes an element in the sequence. In this paper, we call the serialized AST sequence a textual vector. The structural information of ASTs are preserved by the sequence of each element within the textual vector. For this textual vector, we treat it as a semantically meaningful "sentence". The semantic meaning is formed jointly by the elements of the vector and their sequence. Each element within the vector is a semantic unit that helps to form the semantic content. Furthermore, each of the textual elements within the vector is tokenized. Specifically, we map each element in the textual vector to a number. For instance, "int" is mapped to "1", and "param" to "2 ". By doing this, the textual vectors are converted to numeric ones and can be fed to the ML algorithm.

3.2 Deep Feature Model

The reasons for utilizing deep learning to build our feature models are two-fold. First, deep learning based models

1. https://www.checkmarx.com/

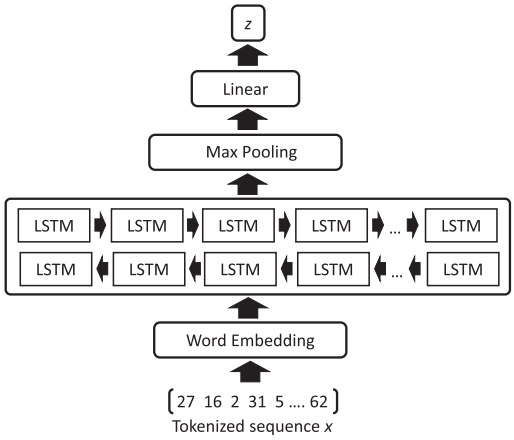


Fig. 4. Architecture of the deep feature model.

achieve state-of-the-art performance on vulnerability detection [15]. Second, deep learning is well known for learning abstract representations of input data. The abstract representations are more likely to be useful in cross-domain applications than the ones including domain dependent details.

The whole architecture of the feature model is illustrated in Fig. 4. It takes the token sequences converted from functions as input. Function names and variable names often differ despite having similar semantics. Similar types should also have similar representations, such as "int", "double", and "float". Thus we map each distinct token into the corresponding token embedding in the same manner as [16]. A token embedding is a continuous vector $\mathbf{e} \in \mathbb{R}^{128}$ and we have $|\mathcal{V}|$ token embeddings, where $\mathcal{V}$ denotes the vocabulary.

The semantics of a function does not only rely on the meaning of individual tokens but also their context. Thus, we feed the embedding sequences into a bidirectional Long Short-Term Memory (BiLSTM) in order to generate more abstract representations. The recurrent neural network LSTM taking the following form:

$$\mathbf{h}_i, \mathbf{c}_i = \mathrm{LSTM}(\mathbf{x}_i, \mathbf{h}_{i-1}, \mathbf{c}_{i-1}), \tag{1}$$

where $\mathbf{x}_i$ is the input at position i, $\mathbf{h}_i \in \mathbb{R}^{64}$ and $\mathbf{c}_i \in \mathbb{R}^{64}$ are the hidden states and cells of LSTM at position i, respectively. The BiLSTM reads a token sequence in two directions, which are an LSTM in the forward direction ($\overrightarrow{\mathbf{h}}_i$) and an LSTM in the backward direction ($\overleftarrow{\mathbf{h}}_i$).

The sequences of $\hat{\mathbf{h}}_i$ are of variable length. The classifier needs an input of fixed length. Thus, we apply max-pooling to take the most significant token features across all tokens in a sequence

$$\bar{\mathbf{h}}_k = \max_{i \in [1,L]} \hat{\mathbf{h}}_{i,k},$$

where $\hat{\mathbf{h}}_{i,k}$ denotes the kth element of $\hat{\mathbf{h}}_i$ at position i, and L is the length of a sequence. After applying max-pooling, we obtain $\bar{\mathbf{h}} \in \mathbb{R}^{128}$ as the hidden representation of a sequence. For even more abstract features, we apply subsequently a linear layer with a weight matrix $\mathbf{W} \in \mathbb{R}^{128 \times 100}$ to generate the final feature vector $\mathbf{z} \in \mathbb{R}^{100}$ of functions.

Pre-Training. Training the feature model is challenging because the model is deep. We found out that layer-wise pre-training [22] can significantly improve the performance. Considering the dataset of some projects is relatively small (i.e., LibTIFF, FFmpeg and LibPNG), we combine them together as input during the pre-training process. We start with training token embeddings by using unlabeled data. Similar to learning word embeddings for natural language processing (NLP) applications, we apply the word2vec[2] toolkit to train token embeddings with the Continuous Bag-of-Words (CBOW) model on the token sequences extracted by CodeSensor. This learning step is unsupervised so we train token embeddings on the token sequences extracted from the union of all projects considered in our experiments. In the next step, we train the feature model with the labeled data in the source domain. We apply a binary logistic regression classifier on top of our feature model. The corresponding loss is cross entropy. We fixed the word embedding to the pre-trained word embeddings during training to reduce over-fitting [23].

3.3 Cross-Domain Representations

At the core of our cross-domain representation learning algorithm is MTLF, which is the state-of-the-art domain adaptation algorithm on multiple benchmark datasets. In our preliminary studies, we have compared several recent DA algorithms. Most of them perform well on images but fail to achieve descent results on text. MTLF is the best performing one on this task. This algorithm assumes that there is a large amount of labeled training data in the source domain, and a handful of labeled data in the target domain. With the help of the labeled data in both domains, MTLF reduces the distribution divergence between $P_s(\mathcal{X})$ and $P_t(\mathcal{X})$ in two ways: i) re-weighting the instances in the source domain; ii) learning Mahalanobis distance to maximize the distance between classes and minimize the distance within each class.

We adapt the algorithm for cross-domain representation learning by feeding the representations $\mathbf{z}$ of token sequences into the training objective of MTLF. The Mahalanobis distance between two token sequence embeddings $\mathbf{z}_i$ and $\mathbf{z}_j$ is defined as

$$d(\mathbf{z}_i, \mathbf{z}_j) = \sqrt{(\mathbf{z}_i - \mathbf{z}_j)^{\mathrm{T}} \mathbf{M} (\mathbf{z}_i, \mathbf{z}_j)},$$

where the matrix $\mathbf{M}$ is positive semi-definite and it can be decomposed as $\mathbf{A}^{\mathrm{T}}\mathbf{A}$. When $\mathbf{M}$ is the identity matrix, it becomes the widely used euclidean distance. Thus it features more capacity than the other euclidean distance based DA methods.

The instance weighting is realized by the function $\omega(\mathbf{z}_i)$. It gives high weights to the instances that are highly correlated to the target domain labeled data. The within-class loss to minimize is defined as

$$l_{\mathrm{in}}(\mathbf{A}, \omega) = \sum_{y_i = y_j} \omega(\mathbf{z}_i)\omega(\mathbf{z}_j) \|\mathbf{A}(\mathbf{z}_i - \mathbf{z}_j)\|^2. \tag{2}$$

The loss is interpreted as minimizing the Mahalanobis distance between any token sequences of the same class. The distance is weighted by the relevance of the involved

2. https://code.google.com/p/word2vec/

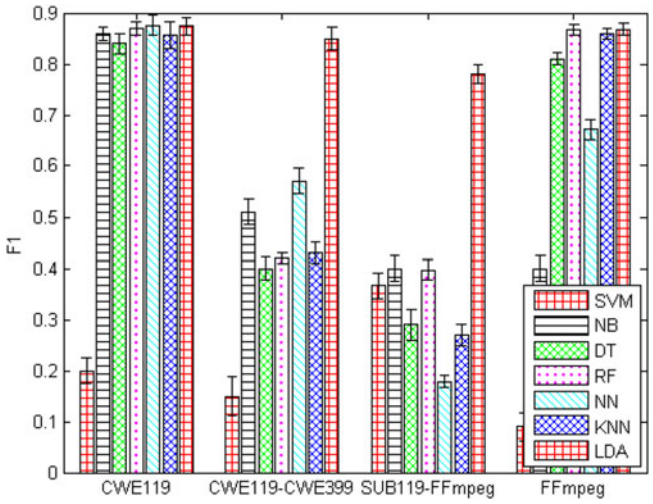


Fig. 5. Classification results based on different classifiers for different code vulnerability scenarios.

instances to the target domain data. The between class loss to maximize has the similar form except the two instances in comparison have different labels

$$l_{\text{out}}(\mathbf{A},\omega)=\sum_{y_i\neq y_j}\omega(\mathbf{z}_i)\omega(\mathbf{z}_j)\|\mathbf{A}(\mathbf{z}_i-\mathbf{z}_j)\|^2. \tag{3}$$

In the MTLF framework, the instance weights are supposed to be close to the density ratio $\omega_0(\mathbf{z}_i)=\frac{P_t(\mathbf{z}_i)}{P_s(\mathbf{z}_i)}$, which are used as a regularizer in the framework. Overall, the training objective of MTLF is formulated as

$$\begin{aligned}\min_{\mathbf{A},\omega}\ &\text{tr}(\mathbf{A}^{\text{T}}\mathbf{A})+\alpha\sum_{i\in\mathcal{D}_s\cup\mathcal{D}_t}\|\omega(\mathbf{z}_i)-\omega_0(\mathbf{z}_i)\|^2\\&+\beta[l_{\text{in}}(\mathbf{A},\omega)-l_{\text{out}}(\mathbf{A},\omega)],\end{aligned} \tag{4}$$

where $\text{tr}(\mathbf{A}^{\text{T}}\mathbf{A})$ is the trace of $\mathbf{M}$ that serves as the regularizer of $\mathbf{A}$, $\mathcal{D}_s$ and $\mathcal{D}_t$ denote the labeled data from the source domain and the target domain respectively, the hyperparameters α, and $\beta\in\mathbb{R}^+$ adjust the importance of the corresponding terms.

The term $\|\mathbf{A}(\mathbf{z}_i-\mathbf{z}_j)\|^2$ in Eqs. (2) and (3) can be rewritten as $\|\mathbf{A}\mathbf{z}_i-\mathbf{A}\mathbf{z}_j\|^2$. This can be considered as applying the transformation matrix $\mathbf{A}$ to project token sequence embeddings $\mathbf{z}_i$ into a new space. In the new space, the data points are supposed to be similar to each other according to euclidean distance if their labels are the same, which is easier for a classifier to learn the decision boundary. In light of this, we apply the matrix $\mathbf{A}$ to all sequence embeddings in both domains to obtain the cross-domain embeddings. It is worth to point out that there is no universal $\mathbf{A}$ for all domains, $\mathbf{A}$ is adaptive only for a pair of source and target domains.

3.4 Classifier

The default classifier of MTLF is the K-nearest neighbour classifier (KNN) [17]. In principle, we can apply any binary classifier on the cross-domain representations of token sequences for vulnerability detection. We have empirically compared a range of classifiers: Linear Discriminant Analysis (LDA) [24], Support Vector Machine (SVM) [4], Naïve Bayes (NB) [25], C4.5 Decision Tree (DT) [25], feedforward neural networks (NN) [25], KNN, and Random Forest (RF) [16], [26]. We run the experiments on the CWE119-CWE399 and SUB119-FFmpeg datasets. SUB199 is a subset of CWE119 that contains randomly sampled 100 vulnerable functions and 300 non-vulnerable functions.

Fig. 5 shows that LDA exhibits outstanding performance on the cross-domain SVD problem. The highest F1 scores of SVM, NB, DT, RF, NN and KNN on both datasets are about 52 and 40 percent, however, the F1 scores obtained by LDA are 85 and 78 percent, which is at least 33 percent higher than the other classifiers. Fig. 5 also demonstrates that LDA can achieve very comparable results with the in-Domain SVD problem. For example, even though RF and NN obtain very high F1 values on the CWE119 dataset, about 87 and 88 percent respectively, the F1 value of LDA is about 87 percent on the CWE119 dataset, which is very comparable with other classifiers. This can be explained by the fact that the cross-domain representations are mostly linear separable. LDA works as a dimensionality reduction algorithm in the linear space. It learns a transformation function $y=\mathbf{w}^{\text{T}}\mathbf{z}$, where $\mathbf{w}$ is a dense vector that transforms the vector $\mathbf{z}$ into a scalar y. The classifier just needs to select an optimal threshold θ that separate data points of two classes in the one-dimensional space. In other words, the advantage of the cross-domain representations simplifies the cross-domain task by projecting data points into a linear separable space.

3.5 Feasibility Analysis

In the previous section, we discussed some preliminary guiding principles for using deep learning and transfer learning for the Cross-Domain SVD problem. Given a representation for a software program (this could be either ASTs or code gadgets), these principles are used to answer questions such as: 1) How to learn features from the program representation? 2) How can we relieve the Cross-Domain SVD problem? To answer these questions, we proposed to use a deep learning algorithm for deep feature representation learning, and we also employed MTLF to leverage the cross-domain problem.

Previous study [1], [15], [27], [28] have shown that deep learning has better performance than traditional machine learning. To evaluate whether deep learning features can improve the classifier's performance. We use two kinds of features, namely the original features and the DL-learned features. The original features refer to the token feature representations that are obtained after embedding, whilst the features learned from DL refer to those that result after the application of BiLSTM to the token features. Our experimental results show that both RF and NN results yield much better performance with the features that are learned from the application of BiLSTM.

Moreover, to show the effectiveness of the MTLF algorithm, we also conducted experiments using SUB119-FFmpeg. Fig. 6 displays the experimental results in terms of traditional machine learning and transfer learning based on SUB119-FFmpeg. One can see that the FMs of RF and NN are about 40 and 17 percent, respectively. However, after we applied MTLF, the FM values increased to more than 50 and 60 percent, rseulting in a significant improvement.

We believe that this is because data domain adaptation reduces the divergence between the training data and test data probability distributions. Thus, the traditional machine

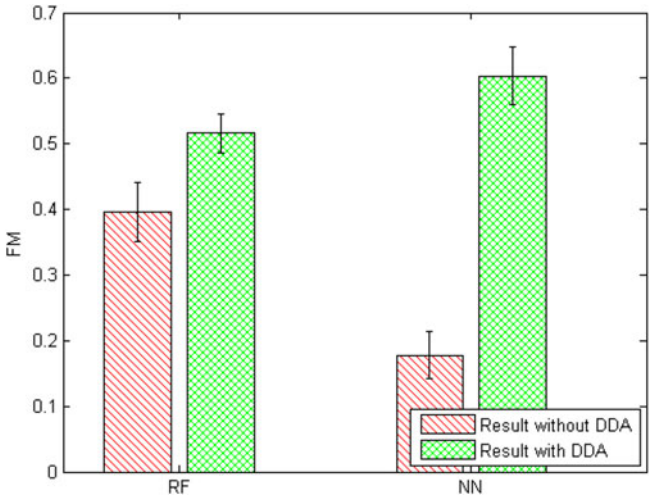


Fig. 6. Classification performance based on two scenarios, Note: DDA ≡ Data Domain Adaptation.

learning algorithms are able to build a robust classification model. This can be seen from Table 1. It is worth noting that the $p-$value has been suggested to be neither as reliable nor as objective as most scientists assume [29], [30]. We employ the Histogram Comparison technique [31] to show whether there are any differences in the data sets after data domain adaptation. We first map each data set into a histogram image, then we calculate the correlation values of the two histograms to determine how well the histograms match each other. We calculate the correlation of the two data sets attribute-by-attribute, then the averaged results are reported to highlight the similarity. The correlation can be calculated as follows:

$$d(H_1, H_2) = \frac{\sum_I (H_1(I) - \bar{H}_1)(H_2(I) - \bar{H}_2)}{\sqrt{\sum_I (H_1(I) - \bar{H}_1)^2 \sum_I (H_2(I) - \bar{H}_2)^2}}, \quad (5)$$

where $\bar{H}_k = \frac{1}{N}\sum_I H_k(I), k = 1, 2.$ and N is the total number of histogram bins. A detailed explanation of Histogram Comparison techniques is beyond the topic of this paper. For more information, please refer to [31].

TABLE 1
Correlation Values of Probability Distribution Comparison for Three Types of Datasets

	CWE399-SUB119	LibPNG-FFmpeg	FFmpeg-CWE399
Original data	0.246	0.169	-0.031
DA Data	**0.766**	**0.683**	**0.950**

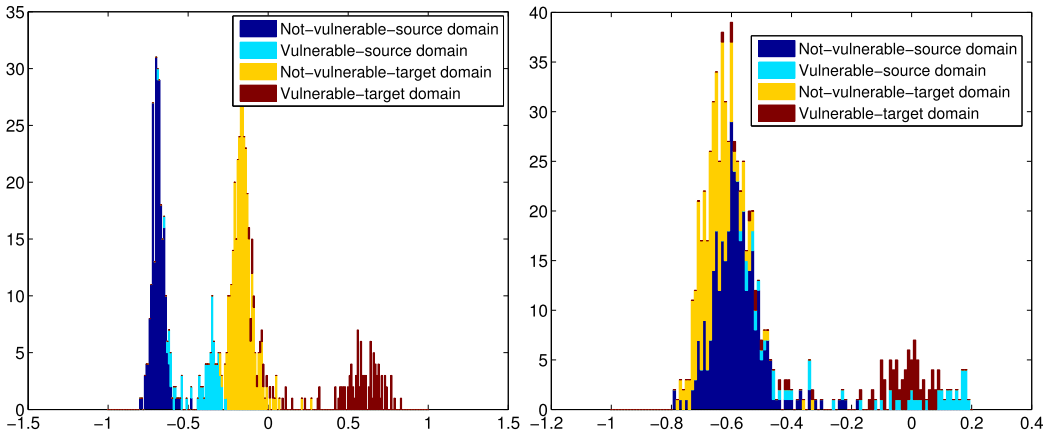
'Original data' refers to the datasets without data domain adaptation, 'DA data' refers to datasets after domain adaptation.

In Table 1, CWE399-SUB119, LibPNG-FFmpeg and FFmpeg-CWE399 represent three scenarios (cross-vulnerability, cross-project and cross-feature domains) of the cross-type SVD problem. As mentioned previously, the larger the correlation value, the better the match. One can see that the correlation values based on the DA technique are at least 0.5 higher than the correlation values obtained based on the original data. This means that DA can help reduce the difference between the training data and test data probability distributions.

In order to further investigate the advantages of cross-domain representation, we visualize both the high-level representations $\mathbf{z}$ of the functions in the FFmpeg dataset and their cross-domain representations (see Fig. 7). Specifically, since LDA learns a function $y = \mathbf{w}^{\mathrm{T}}\mathbf{z}$ that maps each vector $\mathbf{z}$ into a scalar y, we employ an LDA trained in the source domain to project each representation into the one-dimensional space. Fig. 7a illustrates all function representations in both domains. The optimal threshold in the source domain is around -0.5. If we apply this threshold to the target domain data, most of the functions in that domain are classified as vulnerable. In contrast, after transforming the representations $\mathbf{z}$ by the matrix $\mathbf{A}$, all functions are clustered based on their labels so that we can easily draw a decision boundary around -0.3 (Fig. 7b) to separate vulnerable functions from the non-vulnerable ones.

4 Evaluation

Our extensive experiments show that the proposed CD-VulD significantly outperforms the competitive baselines with a wide margin in the settings of cross-vulnerability



(a) In-domain representations. (b) Cross-domain representations.

Fig. 7. Comparison of function representations in FFmpeg before and after domain adaptation.

TABLE 2
Summary of Datasets

Data/project name	# Vuln. functions	# Not Vuln. functions	# Total functions	Feature type
LibTIFF	96	288	384	ASTs
FFmpeg	191	573	764	ASTs
LibPNG	43	129	172	ASTs
LFL-ALL	330	990	1320	ASTs
CWE119	10440	29313	39753	CG
CWE399	7285	14600	21885	CG
CWE-ALL	17725	43913	61638	CG
Asterisk	15	45	60	ASTs
Asterisk	Unclassified		120	ASTs

CG denotes Code gadget.

and cross-project. We also address six research questions to give new insights regarding the new system.

4.1 Experimental Setup

4.1.1 Datasets

Table 2 gives the statistics of all datasets used for training in our experiments. We collected multiple datasets in the C/C++ programming language to cover all our cross-domain settings. We reuse the CWE119 and CWE399 datasets from [15]. Each of them contains only one type of vulnerability, which is buffer errors and resource management errors respectively. We also manually annotated function-level vulnerabilities from three open-source projects : LibTIFF, FFmpeg and LibPNG based on NVD and CVE databases. Excluding all the vulnerable records, we sampled a subset of non-vulnerable functions. Each project contains the same set of nine vulnerability types, which include buffer errors, resource management errors, format string vulnerability, numeric errors, division by zero, input validation, and null-pointer dereference (We follow the vulnerability categorization provided by NVD [16].

In the in-domain setting, to explore the upper bound of the performance by using all the training data, we combine CWE119 and CWE399 to form CWE-ALL, and combine LibTIFF, FFmpeg, and LibPNG to create a dataset called LFL-ALL. In the deep learning process, we use CWE-ALL and LFL-ALL for building the deep learning model, then each dataset will feed to the related model for feature representation learning.

In the cross-domain settings, we split the dataset (i.e., the data with feature representation) in the target domain into three subsets. In the case of the CWE datasets, we used 5 percent of the total data for development, 5 percent of the total as the target domain labeled data, and the remaining as the test set. The proportion for training was kept small because vulnerabilities of new types were often few in the real-world. In the case of the LibTIFF, FFmpeg and LibPNG, we held out one third for development, one third as the labeled data in the target domain, and one third for test. The development set is used to tune hyperparameters. In the in-domain setting, we perform 10-fold cross-validation on each dataset of interest.

4.1.2 Baselines

Although previous work [32] performed an evaluation study regarding the cross-project software vulnerability detection problem, the experimental results are far more acceptable (F-measure ranges from 0.347 to 0.583). We compared MTLF with several recently developed domain adaptation algorithms, and identified that MTLF works well in our scenario. As we discussed in Section 3.3, MTLF will be employed in CD-VulD. As far as we know, no work has been developed for the cross-domain software vulnerability detection problem. The closest works to ours are G-VulD [16] and VulDeePecker [15]. We compare our work with these two recently developed deep learning based systems. To ensure fair comparison, experiments regarding G-VulD + MTLF and VulDeePecker + MTLF have been conducted and reported as well.

4.1.3 Evaluation Metrics

We consider the metrics of Precision, Recall (i.e., TPR), F1-measure (F1), and false positive rate (FPR) to evaluate the proposed approach. (Note that false negative rate (FNR) = 1 - Recall). In order to show the effectiveness of CD-VulD, top-k precision is also considered in this work [27], [28]. In order to measure which approaches performs better when facing class imbalance problem [33], Matthews correlation coefficient(MCC) [34] is also considered in this study. MCC ranges from -1 to 1. 1 means perfect prediction, 0 indicates that the model produces random results, and -1 means inverse prediction.

It is worth to point out that all the vulnerabilities presented in the data sets (Table 2) are trackable/measurable, which is very important for us to evaluate our proposed CD-VulD. For example, given Asterisk project, CD-VulD can be used to predict possible vulnerabilities, then we will look into NVD and CVE databases to check whether the predicted vulnerability is reported or not. The predicted vulnerability will be treated as True Postive (TP) if it was reported in the NVD and CVE databases, otherwise, it will be treated as a False Positive (FP). Moreover, F-measure is related to the minority class (vulnerable class) because we are more interested in the vulnerabilities identified.

4.1.4 Implementation Details

We implement the deep feature model in Keras (2.0.8) with a TensorFlow (1.3.0) as the backend. The token embedding was pre-trained using the Gensim package (3.0.1) with the default settings. We run all experiments on a machine equipped with CentOS Linux 7, two Intel(R) Xeon(R) E5-2690 v3 2.60 GHz CPUs and with 128 GB RAM.

4.2 Results and Discussions

We evaluate our proposed approach through the following six research questions (RQ1-6).

RQ1: Can CD-VulD detect vulnerabilities of new types in the same project? Long-lasting projects often suffer from vulnerabilities of new types as they grow. We conducted experiments on LibTIFF and FFmpeg, which contain vulnerabilities of mixed types and have a sufficient number of vulnerable functions for training. In each project, we consider buffer errors as one domain, and all other types of vulnerabilities as another domain. This is because buffer errors account for roughly 40 percent of the vulnerable functions in each dataset. Precisely, there are 37 buffer errors out of 96 vulnerabilities in

TABLE 3
Results for RQ1, Evaluating Models to Predict Vulnerabilities of New Types in the Same Project

Source-Target	Technique	Evaluation metric				
		Pre	Recall	FPR	F1	MCC
FFmpeg (Buffer-Others)	G-VulD	0.986	0.651	0.003	0.785	0.757
	G-VulD +MTLF	0.944	0.708	0.010	0.809	0.769
	VDP	0.875	0.063	0.003	0.117	0.201
	VDP+ MTLF	0.889	0.242	0.024	0.381	0.418
	CD-VulD	0.983	0.760	0.004	**0.857**	**0.811**
FFmpeg (Others-Buffer)	G-VulD	0.411	0.873	0.417	0.558	0.468
	G-VulD +MTLF	0.500	0.882	0.429	0.638	0.496
	VDP	0.999	0.506	0.000	0.672	0.675
	VDP+ MTLF	0.999	0.556	0.000	0.714	0.693
	CD-VulD	0.999	0.712	0.000	**0.831**	**0.816**
LibTIFF (Buffer-Others)	G-VulD	0.524	0.186	0.056	0.275	0.164
	G-VulD + MTLF	0.571	0.286	0.130	0.381	0.245
	VDP	0.833	0.085	0.006	0.154	0.215
	VDP+ MTLF	0.833	0.167	0.024	0.277	**0.295**
	CD-VulD	0.475	0.458	0.169	**0.464**	0.264
LibTIFF (Others-Buffer)	G-VulD	0.679	0.514	0.081	0.585	0.452
	G-VulD + MTLF	0.714	0.556	0.091	**0.625**	**0.531**
	VDP	1.000	0.027	0.000	0.053	0.151
	VDP+ MTLF	0.909	0.053	0.024	0.099	0.041
	CD-VulD	0.484	0.419	0.149	0.449	0.215

the LibTIFF dataset. There are 79 buffer errors out of 191 vulnerabilities in the FFmpeg dataset. We need to combine the remaining eight types of vulnerabilities, such as numeric errors and input validation errors, to obtain enough functions for training.

From Table 3, we can see that CD-VulD outperforms the other models in three out of four scenarios. All models perform better on FFmpeg than on LibTIFF due to the size of the available training data. On FFmpeg, CD-VulD beats VulDeePecker and VulDeePecker+MTLF with a wide margin in terms of both precision and recall. Our model achieves more than 0.83 F1 scores regardless of which vulnerability types are considered as the target domain. In all of the scenarios, VulDeePecker and VulDeePecker+MTLF achieve high precision but falls short of high recall, which is a clear sign of overfitting. It seems that deep NN models with supervised training tend to memorize too much information of the training data [35]. In the cross-domain setting, G-VulD and G-VulD+MTLF are more robust than VulDeePecker and even performs better than our model when the target domain is buffer errors on LibTIFF project. We conjecture that it might be caused by the use of RF in G-VulD. RF is a robust classifier based on the joint prediction of multiple decision trees capturing different patterns. When the eight vulnerability types are considered as the source domain, there might not be enough data in LibTIFF to train the matrix $\mathbf{A}$.

In addition, one can see that CD-VulD outperforms G-VulD and VulDeePecker on FFmpeg(Buffer-Others), FFmpeg (Others-Buffer) and LibTIFF(Buffer-Others) in terms of MCC. This means that CD-VulD is more robust when facing class imbalance problem. VulDeePecker+MTLF performs better on LibTIFF(Buffer-Others) than CD-VulD because it has higher Precision. However, we can see that the recall of VulDeePecker+MTLF on FFmpeg(Buffer-Others) is only 0.167 which is nearly 20 percent lower than CD-VulD. While G-VulD has higher MCC on LibTIFF(Others-Buffer) compared with CD-VulD, we suspect this because there is lack of vulnerabilities in the target domain (i.e., not enough buffer error samples in LibTIFF to train the matrix $\mathbf{A}$).

RQ2: Can CD-VulD detect vulnerabilities of new types from different projects than the source domain?

Having vulnerable code with both types and projects different than the source domain is a more challenging scenario than RQ1. We consider a relatively simple case by considering only one vulnerability type per project. Thus, we conducted experiments on CWE119 and CWE399 by using each of them in turn as the target domain and the other as the source domain.

From Table 4 one can observe that CD-VulD substantially improves over the baselines regardless of whether CWE119 or CWE399 are used as the target domain. Both precision and recall of CD-VulD go above 0.84 when CWE119 is the source domain and CWE399 is the target domain. In the reverse case, both measures are also more than 0.78. The results suggest that it is more difficult to adapt from resource management errors to buffer errors than the reverse direction. In contrast, G-VulD incurs much higher FPR and lower precision than those of the other competitors regardless of the selection of domains, indicating that G-VulD misclassifies many non-vulnerable functions as vulnerable. In addition, the recall of all the other deep learning-based models are also significantly lower than our model so that they miss a significant amount of vulnerabilities. Surprisingly, VulDeePecker's performance is not much influenced by the choice of domains, given a large amount of training data. Moreover, we can see that both G-VulD+MTLF and VulDeePecker+MTLF perform better than G-VulD and VulDeePecker, which means MTLF can help to reduce the distribution divergence. However, we hypothesis that the high-level representations learned by CD-VulD is linear separable (see Fig. 7), which means LDA performs better than RF and NN. Therefore, CD-VulD outperforms all the other deep learning-based models.

RQ3: Is the performance of CD-VulD robust across different projects, which comprise multiple vulnerability types?

The most challenging scenario is that the source domain and the target domain differ in both projects and vulnerability types, and there are multiple vulnerability types in each project. In this setting, we carried out experiments on the datasets collected from three projects: LibTIFF, LibPNG, and FFmpeg. Each project contains nine types of vulnerability, such as information leak, integer overflow, buffer errors, and renouncement errors. Considering FFmpeg-LibPNG has similar results as LibPNG- FFmpeg, we only report LibPNG- FFmpeg.

As shown in Table 5, CD-VulD is a clear winner over both baselines in terms of precision, recall and F1 in all cases except FFmpeg-LibTIFF. It can achieve F1 around 0.69 when the target domain is LibPNG or FFmpeg. However, the overall performance is lower than the previous scenarios due to smaller training data and mixed types of vulnerability. In contrast, VulDeePecker cannot achieve recall of more

TABLE 4
Results for RQ2

Source-Target	Technique	Evaluation metric			
		Precision	Recall	FPR	F1
CWE119-CWE399	G-VulD	0.352	0.522	0.484	0.419
	G-VulD +MTLF	0.392	0.669	0.510	0.496
	VulDeePecker	0.614	0.533	0.167	0.571
	VulDeePecker +MTLF	0.668	0.814	0.201	0.734
	CD-VulD	0.844	0.865	0.080	**0.854**
CWE399-CWE119	G-VulD	0.405	0.288	0.160	0.330
	G-VulD +MTLF	0.508	0.592	0.204	0.547
	VulDeePecker	0.662	0.674	0.122	0.668
	VulDeePecker +MTLF	0.719	0.670	0.090	0.708
	CD-VulD	0.786	0.786	0.085	**0.777**

Evaluating models to detect vulnerabilities of new types from different projects.

TABLE 5
Results for RQ3

Source-Target	Technique	Evaluation metric				
		Pre	Recall	FPR	F1	MCC
LibTIFF-LibPNG	G-VulD	0.516	0.220	0.061	0.303	0.249
	G-VulD +MTLF	0.785	0.379	0.035	0.512	0.457
	VDP	0.083	0.019	0.032	0.023	0.028
	VDP+ MTLF	0.500	0.034	0.012	0.064	0.076
	CD-VulD	0.772	0.607	0.058	**0.680**	**0.582**
LibTIFF-FFmpeg	G-VulD	0.549	0.414	0.113	0.472	0.343
	G-VulD +MTLF	0.671	0.734	0.120	0.701	0.597
	VDP	0.674	0.022	0.004	0.041	0.130
	VDP+ MTLF	0.857	0.047	0.003	0.089	0.165
	CD-VulD	0.910	0.728	0.024	**0.809**	**0.748**
LibPNG-FFmpeg	G-VulD	0.434	0.111	0.048	0.177	0.111
	G-VulD +MTLF	0.857	0.709	0.039	0.776	0.716
	VDP	0.057	0.044	0.011	0.082	0.105
	VDP+ MTLF	0.696	0.552	0.081	0.615	0.511
	CD-VulD	0.827	0.774	0.054	**0.799**	**0.732**
LibPNG-LibTIFF	G-VulD	0.336	0.135	0.089	0.193	0.063
	G-VulD +MTLF	0.629	0.344	0.068	0.444	0.348
	VDP	0.507	0.099	0.055	0.134	0.139
	VDP+ MTLF	0.358	0.375	0.223	0.366	0.149
	CD-VulD	0.520	0.406	0.125	**0.456**	**0.362**
FFmpeg-LibTIFF	G-VulD	0.586	0.032	0.005	0.060	0.094
	G-VulD +MTLF	0.769	0.156	0.017	**0.260**	0.277
	VDP	0.493	0.057	0.024	0.096	0.091
	VDP+ MTLF	0.714	0.143	0.024	0.238	0.240
	CD-VulD	0.990	0.141	0.000	0.246	**0.307**

Each project is compromised by vulnerabilities of nine types.

than 0.1 in all cases, and thus missed most of the vulnerabilities. It gets also low FPR indicating that it classifies most functions as non-vulnerable. G-VulD works better than VulDeePecker by predicting more vulnerable functions correctly than VulDeePecker, which is similar to the cases in RQ1. We conjecture it may also be caused by the use of the RF algorithm instead of neural networks. Although our model cannot always achieve the lowest FPR, it is just slightly higher than the lowest FPR. G-VulD+MTLF and VulDeePecker+MTLF result in better performance than G-VulD and VulDeePecker, which indicates MTLF can reduce the distribution difference across different projects in terms of multiple vulnerability types. For example, VulDeePecker +MTLF obtains 0.615 F1 measure compared with VulDeePecker with only 0.082 F1 measure because of higher Precision (0.696 versus 0.057 of VulDeePecker) and Recall (0.552 versus 0.044 of VulDeePecker). This means VulDeePecker +MTLF can detect more vulnerabilities than VulDeePecker. However, CD-VulD outperforms G-VulD+MTLF and VulDeePecker+MTLF with a large margin. We believe this is because the high-level representations learned by CD-VulD is linear separable, LDA employed by CD-VulD outperforms RF and NN in such scenario. In addition, Table 5 shows that CD-VulD has the best MCC among all the cases, which indicates CD-VulD is more robust when facing the class imbalance problem. Considering the high precision and recall it gains, the predictions of our model are certainly much more reliable than the competitors.

From Table 5, we also observe that all three systems perform poorly on the scenario of FFmpeg-LibTIFF, even though CD-VulD performs better than the other two approaches. We conjecture that the size of the LibTIFF dataset might be too small. In our experiments, we select one-third of the samples from the LibTIFF dataset for distance metric learning. Therefore, the target domain labeled data might not be enough for building a robust classification model.

RQ4: Is the performance of CD-VulD invariant to the syntactic structures of input? This question is important because the source domain and target domain might have different syntactic structures. For example, code gadgets are available for CWE119 and CWE399, however, they are not available for other projects such as FFmpeg and LibTIFF (refer to Section 3.1.2). One might want to consider CWE119 as source domain given that there are not enough training samples with FFmpeg and LibTIFF.

To show the performance of CD-VulD is robust regardless of whether token sequences are constructed from ASTs or code gadgets, we run experiments on the datasets: CWE119, CWE399, FFmpeg, and LibTIFF. CWE119 and CWE399 are based on code gadgets, while the syntactic structure of FFmpeg and LibTIFF are ASTs. Note that, there are some vulnerabilities (buffer errors and management errors) overlap between CWE119, CWE399 and FFmpeg but there are no overlap between CWE119, CWE399 and LibTIFF. We choose FFmpeg and LibTIFF not only to evaluate the cross-domain scenario in different feature spaces but also the cross-domain where there may existing overlap of history vulnerabilities. This because code reuse/clone is a common practice in the software industry and vulnerability code can be copied and reused by code clone. Considering CWE119-LibTIFF has similar results as LibTIFF- CWE119, and CWE339-LibTIFF has similar results as LibTIFF-CWE339, we report LibTIFF- CWE119 and LibTIFF-CWE339 as the representations.

As shown in Table 6, CD-VulD has the highest Precision and Recall, and lowest FPR, among most of the systems in comparison. Comparing with the experimental results based on the same syntactic structures (RQ1-RQ3), the F1 scores of CD-VulD are at a similar level, regardless of the

TABLE 6
Results for RQ4

Source-Target	Technique	Evaluation metric			
		Precision	Recall	FPR	F1
CWE119-FFmpeg	G-VulD	0.965	0.440	0.005	0.604
	G-VulD +MTLF	0.919	0.532	0.016	0.674
	VulDeePecker	0.085	0.277	0.998	0.130
	VulDeePecker +MTLF	0.632	0.319	0.062	0.425
	CD-VulD	0.992	0.663	0.002	**0.795**
FFmpeg - CWE119	G-VulD	0.250	0.996	0.972	0.406
	G-VulD +MTLF	0.713	0.420	0.060	0.528
	VulDeePecker	0.311	0.073	0.023	0.049
	VulDeePecker +MTLF	0.669	0.461	0.034	0.450
	CD-VulD	0.662	0.658	0.142	**0.649**
CWE399-FFmpeg	G-VulD	0.165	0.592	0.998	0.258
	G-VulD +MTLF	0.988	0.512	0.002	0.674
	VulDeePecker	0.089	0.293	0.998	0.137
	VulDeePecker +MTLF	0.821	0.561	0.041	0.667
	CD-VulD	0.898	0.732	0.003	**0.842**
FFmpeg - CWE399	G-VulD	0.240	0.946	0.999	0.382
	G-VulD +MTLF	0.598	0.578	0.130	0.588
	VulDeePecker	0.464	0.066	0.054	0.096
	VulDeePecker +MTLF	0.785	0.244	0.022	0.373
	CD-VulD	0.917	0.920	0.042	**0.918**
LibTIFF-CWE119	G-VulD	0.264	0.751	0.726	0.386
	G-VulD +MTLF	0.649	0.411	0.074	0.503
	VulDeePecker	0.232	0.019	0.090	0.027
	VulDeePecker +MTLF	0.533	0.267	0.078	0.356
	CD-VulD	0.893	0.782	0.038	**0.828**
LibTIFF-CWE399	G-VulD	0.730	0.422	0.052	0.535
	G-VulD +MTLF	0.924	0.544	0.015	0.685
	VulDeePecker	0.482	0.622	0.222	0.543
	VulDeePecker +MTLF	0.623	0.733	0.148	0.673
	CD-VulD	0.625	0.821	0.165	**0.710**

The source domain and the target domain differ not only in projects, vulnerability types but also the syntactic structures of input.

choice of the source domain and the target domain. The change of syntactic structures across domains appear to have little impact on the performance of our model.

The low precision or recall of G-VulD and VulDeePecker indicates that they are sensitive to the order of tokens, as well as the choice of token subsets. This is in particular true for VulDeePecker, which tends to memorize too much from the source domain data. Thus it suffers from the lowest performance in the target domain. In contrast, the cross-domain representation of CD-VulD is much more invariant to the change of order and the selection of tokens.

Therefore, we empysize that CD-VulD can identify more vulnerabilities compared with G-VulD and VulDeePecker in terms of different syntactic structures in source domain and target domain.

RQ5: Can CD-VulD detect recent vulnerabilities? As the test sets in the target domain, we use two kinds of emerging functions: i) recently published vulnerabilities in the public database; ii) new functions randomly sampled from the web.

Recently Published Vulnerabilities. This set of experiments is to verify that CD-VulD can identify recently reported vulnerabilities by training only on historical data. As the test set, we have collected 13 vulnerabilities from the project FFpmeg, which are published in NVD and CVE in 2017, while the most recent vulnerability in our training data is from 2016. Among the 13 vulnerabilities, 8 of them are resource management errors and 5 of them are buffer errors. For training, we consider CWE399 as the source domain data, and FFmpeg as the target domain labeled data. These datasets are chosen to simulate the real-world scenario that there is a large dataset available in the source domain and there is a small labeled data in the target domain with mixed vulnerability types.

From Table 7, one can see that CD-VulD can recognize all the resource management error vulnerabilities with confidence ranging from 0.5383 to 0.8936 (confidence > 0.5 indicates the function is probably vulnerable, higher confidence means the function has high probability of being vulnerable). Although the classifier fails to recognize a buffer error with confidence $0.4781 < 0.5$, however, it can identify the other four vulnerable functions with high confidences, which are more than 0.7. It is worth noting that the rationale behind the fluctuating confidence is that the test cases are different in terms of code content and code length. In this case, the learned features can be different because CD-VulD extracts and learns high-representation features automatically. Comparatively, G-VulD can identify 5 vulnerabilities including 2 management errors and 3 buffer errors, out of 13 vulnerabilities. While VulDeePecker can detect 3 management errors. Therefore, G-VulD and VulDeePecker cause 8 false negatives and 10 false negatives, respectively.

Randomly Sampled Functions. In order to verify if CD-VulD trained on the historical data is able to identify vulnerabilities from a mixture of vulnerable and non-vulnerable functions. We collected and manually labeled 15 vulnerable functions and 45 non-vulnerable functions in the Asterisk project as the target domain labeled data. Then, we collected another 120 test functions from the Asterisk project.

TABLE 7
Results for RQ5

	Tech.	Vulnerable CVE ID												
		CVE-2017-14054	CVE-2017-14055	CVE-2017-14056	CVE-2017-14057	CVE-2017-14058	CVE-2017-14059	CVE-2017-14170	CVE-2017-14171	CVE-2017-15672	CVE-2017-16840-1	CVE-2017-16840-2	CVE-2017-16840-3	CVE-2017-17081
Conf.	G-VulD	0.4458	0.5025	0.4806	0.5608	0.4052	0.3675	0.4066	0.4835	0.4828	0.6469	0.5649	0.4886	0.6251
	VDP	0.5403	0.5442	0.4112	0.4313	0.3716	0.5329	0.4213	0.3626	0.0115	0.0290	0.4884	0.4555	0.1487
	CD-VulD	0.5383	0.5406	0.7318	0.8210	0.8936	0.5404	0.6711	0.7458	0.4781	0.8343	0.7724	0.8177	0.9440

The confidence of CD-VulD on CWE399-FFmpeg for the thirteen vulnerable function (8 management error and 5 buffer errors) detection with confidence > 0.5 (except CVE2017-15672). This indicates that CD-VulD can make use of Off-the-shelf data for unclassified/unknown vulnerable function identification. Conf.: confidence. VDP: VulDeePecker.

TABLE 8
Results for RQ5

Source-Target	TOP5 Precision	TOP10 Precision	TOP20 Precision	TOP30 Precision
LibTIFF-Asterisk	80%	80%	65%	50%
FFmpeg-Asterisk	80%	80%	65%	43%
LibPNG -Asterisk	80%	80%	65%	56%
CWE119-Asterisk	80%	80%	65%	50%

This is the application of CD-VulD in real-world unclassified data. The experimental results indicate that CD-VulD is applicable and practical when facing unclassified real-world data.

Asterisk is a dataset that has only 60 functions labeled as vulnerable, while the remaining functions may be either vulnerable or non-vulnerable. Instead of manually annotating all functions in the test set, we only checked the top-K instances based on $P(y = 1|\mathbf{x})$ of the LDA classifier. Table 8 reports the results of selecting LibTIFF, FFmpeg, LibPNG, and CWE119 as the source domain data. From the results, we can see that the top 5, top 10, and top 20 predictions of CD-VulD achieves 80, 80 and 65 percent precision respectively.

In order to further challenge the proposed system, we manually identified another 10 functions with buffer errors from Asterisk. Since CWE119 corresponds to buffer errors, we use CWE119 as the source domain data, and reuse the target domain labeled data from the Asterisk project. Table 9 shows the confidences of the LDA classifier of CD-VulD regarding each function compromised by buffer errors. We can see that CD-VulD successfully identifies all sampled vulnerabilities by having the confidences larger than 0.5.

RQ6: Can CD-VulD achieve competitive results for SVD in the in-Domain scenario? The research question is to verify if CD-VulD is on par with competitors in the in-domain setting. Thus, we performed cross-validation on each of the seven datasets: LibTIFF, LibPNG, FFmpeg, CWE119, CWE399, LFL-ALL, and CWE-ALL. Since there is no distinction between domains, we do not need to learn the cross-domain representations, but only apply the pre-trained deep feature model to train the LDA classifier.

From Table 10, one can see that CD-VulD achieves competitive results in finding vulnerabilities compared to G-VulD and VulDeePecker. CD-VulD outperforms the competitors on the LibTIFF and CWE119 datasets. The precision of G-VulD on LibTIFF is 0.757, which is about 4 percent higher than CD-VulD, however, its recall (0.445) and F1 (0.541) are at least 8 percent lower than CD-VulD. We observe the similar performance of VulDeePecker on CWE119. Although G-VulD achieves the best performance on LibPNG and FFmpeg, the F1 scores of CD-VulD are comparable, with a shortfall of only 0.062 and 0.003 respectively. Overall, it is evident that CD-VulD performs well for in-domain vulnerability detection.

TABLE 10
Results for RQ6

Data	Technique	Performance metric			
		Precision	Recall	FPR	F1
LibTIFF	G-VulD	0.757	0.445	0.054	0.541
	VulDeePecker	0.500	0.600	0.2069	0.545
	CD-VulD	0.714	0.555	0.069	**0.625**
LibPNG	G-VulD	0.889	0.897	0.043	**0.882**
	VulDeePecker	0.790	0.873	0.093	0.808
	CD-VulD	0.760	0.900	0.108	0.820
FFmpeg	G-VulD	0.957	0.801	0.012	**0.870**
	VulDeePecker	0.829	0.630	0.025	0.672
	CD-VulD	0.975	0.787	0.007	0.867
CWE119	G-VulD	0.906	0.836	0.031	0.869
	VulDeePecker	0.921	0.825	0.025	0.870
	CD-VulD	0.896	0.853	0.035	**0.874**
CWE399	G-VulD	0.974	0.966	0.013	**0.970**
	VulDeePecker	0.972	0.943	0.014	0.947
	CD-VulD	0.975	0.964	0.012	**0.970**
LFL-ALL	G-VulD	0.974	0.196	0.002	0.320
	VulDeePecker	0.807	0.538	0.049	0.590
	CD-VulD	0.879	0.515	0.024	**0.646**
CWE-ALL	G-VulD	0.935	0.889	0.025	0.911
	VulDeePecker	0.943	0.884	0.022	**0.913**
	CD-VulD	0.930	0.898	0.027	**0.913**

All systems are compared in the in-domain setting.

5 Limitations

Our CD-VulD work has several limitations. We plan to address the following issues of CD-VulD in future work.

First, CD-VulD focuses on detecting vulnerabilities in source code. However, there are also a large number of vulnerabilities present in binary code for which no source code is available, and so it would be interesting to see if CD-VulD can be adapted to binary code.

Second, the evaluation of CD-VulD in this paper is limited to nine vulnerability types and most of these types can only be found in the four small projects (LibTIFF, LibPNG, FFmpeg, and Asterisk). We plan to collect more datasets with additional vulnerability types and test our CD-VulD system on those datasets.

Third, although we empirically choose LDA as the best performing classifier amongst all candidate classifiers, it remains unclear if the cross-domain representations of all vulnerability types are linearly separable. If not, it would be interesting to explore non-linear classifiers on top of those representations.

TABLE 9
Results for RQ5

	Vulnerable CVE ID									
	CVE-2011-0495	CVE-2012-1183	CVE-2012-1184	CVE-2013-7100	CVE-2017-7617	CVE-2011-2529	CVE-2012-2415	CVE-2012-2416	CVE-2013-2685	CVE-2013-2686
Confidence	0.7549	0.6999	0.7433	0.7182	0.7588	0.6972	0.7238	0.7203	0.7359	0.7596

The confidence of CD-VulD trained on CWE119-Asterisk for the ten vulnerable buffer overflow errors detection with $confidence > 0.5$. This indicates that CD-VulD can make use of Off-the-shelf data for unclassified/unknown vulnerable function identification.

Fourth, CD-VulD is currently limited to employing a Recurrent Neural Network (RNN)-based architecture and domain adaptation. We plan to collect more ground truth data and compare CD-VulD with Convolutional Neural Network (CNN)-based approaches.

Fifthly, in our study, we identified the recently developed domain adaptation algorithm, called MTLF, that performs well in our scenario. We are also interested in deep learning-based domain adaptation techniques. We hypothesise that deep learning-based domain adaptation techniques may be able to discover more software vulnerabilities.

Finally, in our study, the experiments conducted are based on C/C++ programming languages. We plan to apply and evaluate our proposed method to other programming languages such as Java and C# in the future.

6 Related Work

Software vulnerability detection has been a fundamental problem in the field of cyber-security and received much attention from the research community. Our discussion of related work mainly focuses on approaches that apply ML techniques for detecting software vulnerabilities [3].

Many works have been proposed to prevent vulnerabilities at both binary code level and source code level [27], [28]. Morrison *et al.* [36] experimentally demonstrated that source code file level prediction is preferred by engineers because the binaries are usually too large to be used for practical inspection. In Shar and Tan's work [25], a set of static code attributes that characterise such code patterns have been proposed. Yamaguchi *et al.* [37] explored the taint-style vulnreabilities and proposed the automatic inference of search patterns based on C source code. Alves *et al.* [5] studied 18 state-of-the-art approaches based on a larege data set. Their experiments showed that RF performs the best. Eschweiler *et al.* [38] developed a new method for function similarity identification based on control flaw graph of binary code. Grieco *et al.* [26] presented a scalable machine learning approach for vulnerability detection based on lightweight static and dynamic features. They also developed an open source tool, VDiscover. The took makes use of the state-of-the-art ML techniques for software vulnerability detection. Liu *et al.* [28] combine deep learning and a fuzzy-based over sampling for software vulnerability detection.

Apart from this, the choice of features plays an important role in software vulnerability detection. It not only can affects the detection capability but also can affect the detection granularity. Neuhaus *et al.* [39] used features extracted from imports and functions calls and trained an SVM classifier for predicting vulnerabilities in Mozilla software. Perl *et al.* [4] presented VCCFinder which employs SVMs for potential vulnerability detection. Scandariato *et al.* [40] proposed to analyze the source code directly and make use of the bag-of-words (BoW) for feature representation learning. Hoa *et al.* [41] proposed to leverage the deep learing-based approach to learn high feature representations from both semantic features and syntactic features. Their experiments showed much improvement for within-project and cross-project. The fragmentation of developer contributions were first considered by Pinzger *et al.* [42] to predict the number of post-release failures. Later, Meneely and Williams [43] explored the correlations between the known security vulnerabilities and the developer activity. Their study showed that some of the features selected by knowledgeable domain experts may carry outdated experience and underlying biases, and the features that perform well in one project may not perform well in other projects. [44].

The most closely related work to this study are Lin *et al.* [16] (referred to as G-VulD) and Li *et al.* [15] (referred to as VulDeePecker). G-VulD employs deep learning for function-level vulnerability detection by using unlabeled data, while VulDeePecker is a deep learning-based system for vulnerability detection. There are several differences between our work and other works. First, CD-VulD is developed for the scenario of cross-domain software vulnerability detection, while G-VulD and VulDeePecker were developed for within the domain scenario based on ASTs and Code Gadgets. CD-VulD employs MTLF to minimize the distribution divergence between the source domain and the target domain. However, even if MTLF is employed in G-VulD and VulDeePecker, our experiments show that CD-VulD still achieves outstanding performances; second, CD-VulD works on within-domain as well. Our experimental results show that CD-VulD achieves very comparative results with G-VulD and VulDeePecker for within-domain scenario. However, G-VulD and VulDeePecker have poor performance in the cross-domain scenario; third, CD-VulD is a scalable approach. CD-VulD can be applied to different features. However, VulDeePecker and G-VulD are developed for code gadget and ASTs features, respectively; finally, CD-VulD employs LDA because we identified the high-representation features are linear separable, while VulDeePecker and G-VulD use RF and NN, respectively.

7 Conclusion

In this paper, we proposed CD-VulD, which is the first approach for cross-domain software vulnerability detection based on deep learning. At the core of CD-VulD is a deep feature model and a method to learn cross-domain representations. The feature model projects token sequences converted from ASTs into in-domain high-level representations. We map the high-level representations into the cross-domain representations using a transformation matrix learned by using MTLF. Those representations are then used as input to an LDA classifier for vulnerability detection. We have demonstrated by means of extensive experiments that CD-VulD achieves a superior performance compared with baselines in a range of cross-domain settings, including cross-project, cross-vulnerability, and prediction of recent software vulnerabilities - though the baselines are not designed for cross-domain vulnerability detection. The deep feature model also achieved comparative results compared with the baselines when in the in-domain setting.

Acknowledgments

This research was supported under the Defence Science and Technology Group's Next Generation Technologies Program, the Discovery Project under contract no. DP200100886 and the Linkage Project under Grant LP180100170.

REFERENCES

[1] X. Ban, S. Liu, C. Chen, and C. Chua, "A performance evaluation of deep-learnt features for software vulnerability detection," *Concurrency Comput., Practice Experience*, vol. 31, no. 19, 2019, Art. no. e5103.

[2] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: A scalable approach for vulnerable code clone discovery," in *Proc. IEEE Symp. Security Privacy*, 2017, pp. 595–614.

[3] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey," *ACM Comput. Surv.*, vol. 50, no. 4, 2017, Art. no. 56.

[4] H. Perl *et al.*, "VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 426–437.

[5] H. Alves, B. Fonseca, and N. Antunes, "Experimenting machine learning techniques to predict vulnerabilities," in *Proc. 7th Latin-Amer. Symp. Dependable Comput.*, 2016, pp. 151–156.

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

[7] L. Liu, O. De Vel , Q.-L. Han, J. Zhang, and Y. Xiang, "Detecting and preventing cyber insider threats: A survey," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 2, pp. 1397–1417, Second Quarter 2018.

[8] R. Coulter, Q.-L. Han, L. Pan, J. Zhang, and Y. Xiang, "Data-driven cyber security in perspective–intelligent traffic analysis," *IEEE Trans. Cybern.*, to be published, doi: 10.1109/TCYB.2019.2940940.

[9] J. Jang, A. Agrawal, and D. Brumley, "ReDeBug: Finding unpatched code clones in entire os distributions," in *Proc. IEEE Symp. Security Privacy*, 2012, pp. 48–62.

[10] C. Cadar *et al.*, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Conf. Operating Syst. Des. Implementation*, 2008, pp. 209–224.

[11] 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," *ACM SIGOPS Operating Syst. Rev.*, vol. 35, pp. 57–72, 2001.

[12] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. London, U.K.: Pearson, 2007.

[13] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," 2005.

[14] M. D. Ernst, "Static and dynamic analysis: Synergy and duality," in *Proc. ICSE Workshop Dyn. Anal.*, 2003, pp. 24–27.

[15] Z. Li *et al.*, "VulDeePecker: A deep learning-based system for vulnerability detection," in *Proc. 25th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2018.

[16] G. Lin, J. Zhang, W. Luo, L. Pan, and Y. Xiang, "POSTER: Vulnerability discovery with function representation learning from unlabeled projects," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 2539–2541.

[17] J. Zhang, Y. Xiang, Y. Wang, W. Zhou, Y. Xiang, and Y. Guan, "Network traffic classification using correlation information," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 1, pp. 104–117, Jan. 2013.

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

[19] X. Chen *et al.* "Android HIV: A study of repackaging malware for evading machine-learning detection," *IEEE Trans. Inf. Forensics Security*, vol. 15, pp. 987–1001, 2019.

[20] Y. Xu *et al.*"A unified framework for metric transfer learning," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 6, pp. 1158–1171, Jun. 2017.

[21] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proc. 28th Annu. Comput. Secur. Appl. Conf.*, 2012, pp. 359–368.

[22] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2007, pp. 153–160.

[23] Y. Qian, Y. Fan, W. Hu, and F. K. Soong, "On the training aspects of deep neural network (DNN) for parametric TTS synthesis," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2014, pp. 3829–3833.

[24] S. S. Murtaza, W. Khreich, A. Hamou-Lhadj , and A. B. Bener, "Mining trends and patterns of software vulnerabilities," *J. Syst. Softw.*, vol. 117, pp. 218–228, 2016.

[25] L. K. Shar and H. B. K. Tan, "Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns," *Inf. Softw. Technol.*, vol. 55, no. 10, pp. 1767–1780, 2013.

[26] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," in *Proc. 6th ACM Conf. Data Appl. Secur. Privacy*, 2016, pp. 85–96.

[27] G. Lin *et al.*, "Software vulnerability discovery via learning multi-domain knowledge bases," *IEEE Trans. Dependable Secure Comput.*, to be published, doi: 10.1109/TDSC.2019.2954088.

[28] S. Liu, G. Lin, Q.-L. Han, S. Wen, J. Zhang, and Y. Xiang, "DeepBalance: Deep-learning and fuzzy oversampling for vulnerability detection," *IEEE Trans. Fuzzy Syst.*, to be published, doi: 10.1109/TFUZZ.2019.2958558.

[29] R. Nuzzo, "Scientific method: Statistical errors," *Nat. News*, vol. 506, no. 7487, 2014, Art. no. 150.

[30] M. Baker *et al.*, "Statisticians issue warning on P values," *Nature*, vol. 531, no. 7593, 2016, Art. no. 151.

[31] H. Ling and K. Okada, "Diffusion distance for histogram comparison," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, 2006, pp. 246–253.

[32] A. Webster, "A comparison of transfer learning algorithms for defect and vulnerability detection," *Digit. Repository Univ. Maryland*, pp. 1–11, 2017.

[33] S. Liu, Y. Wang, J. Zhang, C. Chen, and Y. Xiang, "Addressing the class imbalance problem in Twitter spam detection using ensemble learning," *Comput. Secur.*, vol. 69, pp. 35–49, 2017.

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

[35] C. Song, T. Ristenpart, and V. Shmatikov, "Machine learning models that remember too much," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 587–601.

[36] P. Morrison, K. Herzig, B. Murphy, and L. Williams, "Challenges with applying vulnerability prediction models," in *Proc. Symp. Bootcamp Sci. Secur.*, 2015, Art. no. 4.

[37] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," in *Proc. IEEE Symp. Security Privacy*, 2015, pp. 797–812.

[38] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla , "Discovre: Efficient cross-architecture identification of bugs in binary code," in *Proc. 23rd Annu. Netw. Distrib. Syst. Secur. Symp.*, 2016.

[39] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proc. 14th ACM Conf. Comput. Commun. Secur.*, 2007, pp. 529–540.

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

[41] H. K. Dam, T. Tran, T. T. M. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for predicting vulnerable software components," *IEEE Trans. Softw. Eng.*, to be published, doi: 10.1109/TSE.2018.2881961.

[42] M. Pinzger, N. Nagappan, and B. Murphy, "Can developer-module networks predict failures?" in *Proc. 16th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2008, pp. 2–12.

[43] A. Meneely and L. Williams, "Secure open source collaboration: An empirical study of Linus' Law," in *Proc. 16th ACM Conf. Comput. Commun. Secur.*, 2009, pp. 453–462.

[44] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: A large scale experiment on data vs. domain vs. process," in *Proc. 7th Joint Meet. Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2009, pp. 91–100.