Wireless Pers Commun (2018) 102:2261–2285
https://doi.org/10.1007/s11277-017-5069-3

Defect Prediction in Android Binary Executables Using Deep Neural Network

Feng Dong[1] · Junfeng Wang[2] · Qi Li[1] · Guoai Xu[1] · Shaodong Zhang[1]

Published online: 21 November 2017

Abstract Software defect prediction locates defective code to help developers improve the security of software. However, existing studies on software defect prediction are mostly limited to the source code. Defect prediction for Android binary executables (called apks) has never been explored in previous studies. In this paper, we propose an explorative study of defect prediction in Android apks. We first propose smali2vec, a new approach to generate features that capture the characteristics of smali (decompiled files of apks) files in apks. Smali2vec extracts both token and semantic features of the defective files in apks and such comprehensive features are needed for building accurate prediction models. Then we leverage deep neural network (DNN), which is one of the most common architecture of deep learning networks, to train and build the defect prediction model in order to achieve accuracy. We apply our defect prediction model to more than 90,000 smali files from 50 Android apks and the results show that our model could achieve an AUC (the area under the receiver operating characteristic curve) of 85.98% and it is capable of predicting defects in apks. Furthermore, the DNN is proved to have a better performance than the traditional shallow machine learning algorithms (e.g., support vector machine and naive bayes) used in previous studies. The model has been used in our practical work and helped locate many defective files in apks.

✉ Junfeng Wang
wangjf@scu.edu.cn

Feng Dong
dongfeng@bupt.edu.cn

Qi Li
liqi2001@bupt.edu.cn

Guoai Xu
xga@bupt.edu.cn

Shaodong Zhang
zhangsd@bupt.edu.cn

[1] Beijing University of Posts and Telecommunications, Beijing, China

[2] Sichuan University, Chengdu, China

Keywords Software defect prediction · Mobile security · Android binary executables · Machine learning · Deep neural network

1 Introduction

A software defect is an error or a bug built into the software during software design and development on account of a programmer's mistake, such as deviations from the pre-concerted running path, memory overflow, and runtime exceptions [17]. These defects can trigger serious security issues, especially in business with low fault-tolerant rate, such as financial services and electric-power industries. For instance, the so-called Heartbleed Bug released in 2015[1] affected the entire software industry and exposed encryption systems to attack.

Numerous software defect prediction models have been proposed [17]. Prediction models extract defect features from known software to serve as inputs and use machine learning (ML) algorithms for training. They then use the model to predict whether a new software is defective. Research in the field of software defect prediction mainly focuses on two main points: (1) extraction of defect features, and (2) developing more efficient ML algorithms to improve the accuracy. In terms of feature extraction, various features are extracted to distinguish defective modules from normal ones, such as features based on object-oriented software metrics, text features based on string text, and semantic features based on the abstract syntax tree (AST), program dependence graph (PDG), control flow graph (CFG), and others [20, 25]. For ML methods, various traditional algorithms have been used in model training [15, 16, 26] and defect predicting, such as Support Vector Machine (SVM), Naive Bayes (NB), Decision Tree (DT), Logistic Regression (LR), Random Forest (RF) and Neural Network (NN) [12, 13, 22] and so on.

However, existing software prediction models are mostly limited to the source code, but the files obtained in prediction work are usually binary executables. The defect prediction research community and third-party security companies are difficult to get the source code due to software copyright and source code protection restrictions. The prediction for Android binary executables has never been explored in previous studies. Moreover, traditional machine learning algorithms such as SVM, LR, NB have a limited generalization capability for complicated classification tasks because of a shallow architecture. DNN, which has a deep architecture with multiple hidden layers, is able to extract better features and model more complex data than shallow algorithms using the intermediate hidden layers [1].

In this paper, we design an accurate defect prediction model for Android apks. We first propose ***smali2vec*** to generate features that capture the characteristics of defective samli files in apks. Smali2vec generates the comprehensive features which capture both token and semantic features of defects in smali files of the apks. We classify the instructions into eight categories and build the dalvik instruction mapping table for the generation of token vectors and semantic vectors.

Then we leverage DNN instead of the traditional shallow machine learning algorithms to train and build a more accurate defect prediction model. After the model is fully trained, it can locate defective smali files in apks without the source code. We apply the prediction model to a large dataset containing 92,220 smali files from 50 apks (ten Android projects, each in five versions). Like most defect prediction studies [25, 29], experiments are

[1] Heartbleed, https://en.wikipedia.org/wiki/Heartbleed.

conducted in within-project defect prediction (WPDP) mode and cross-project defect prediction (CPDP) mode respectively. The result of the prediction model is evaluated by the receiver operating characteristic and area under the curve (ROC-AUC) evaluation system.

The following research questions are addressed in this work:

1. *RQ1: Is the defect prediction model capable of predicting defects in apks and what is the best configuration of the DNN classifier?* We design ten sets of within-project defect prediction experiments with different parameters of DNN. The experimental results show the ability of predicting defects of the model and the most optimum parameters.
2. *RQ2: Does the comprehensive features based on the mapping table outperform the full-opcode features?* We compare the accuracy of the models generated by the comprehensive features based on the mapping table and the full-opcode features.
3. *RQ3: Does the DNN algorithm outperform the traditional ML algorithms in WPDP mode?* We conduct ten sets of WPDP experiments with DNN and the traditional ML algorithms. The performance of different algorithms is compared by the AUC values.
4. *RQ4: Does the proposed DNN algorithm outperform the traditional ML algorithms in CPDP mode?* Similar to RQ2, we compare the performance of different algorithms in cross-project defect prediction.
5. *RQ5: What is the time and space cost of the DNN algorithm versus traditional ML algorithms?* We compare the time and space costs of different algorithms.

In general, the research presented in this paper makes significant contributions for Android defect prediction as follows:

1. To the best of our knowledge, we take the first step to build the defect prediction model for apks.
2. We propose smali2vec, a new approach to generate features that capture the characteristics of defective files in apks
3. We adapt DNN for defect prediction instead of the traditional ML algorithms. Our evaluation results shows that the DNN algorithm outperforms the other algorithms in both WPDP and CPDP mode.

The rest of this paper is organized as follows. Section 2 provides the related work about software defect prediction. Section 3 describes the methodology of the key steps of our proposed defect prediction model in this paper. Section 4 shows the experimental setup. Section 5 evaluates the performance of our model and answers the research questions. The possible limitations of our model are presented in Sect. 6 and the conclusions are presented in Sect. 7.

2 Related Work

2.1 Software Defect Prediction

A software defect is an error or a bug built into the software during software design and development on account of a programmer error [17]. For example, a programmer may create an unreleased resource defect if he/she forgets to release a socket after communication ends. This can result in a software reliability problem when a large number of sockets are established. Common software defects include unreleased resources, side

channel data leakage, un-normalized input strings, privacy violations, and so on.[2] Figure 1 depicts the classic software defect prediction procedure. First, features that capture the characteristics of software defects are extracted from software components (files, version management tools, methods, etc.) [8]. Then, components are labeled as defective or normal according to the existence of errors or bugs. The third step is to train the ML model with the labeled data. Finally, an unlabeled component is input into the trained model to predict whether it is defective or not. The predicting procedure is regarded as WPDP mode when training and testing sets are derived from the same project. Otherwise, it is regarded as CPDP mode. In our model, files are viewed as components and predictions are made in the two above prediction modes. Table 1 outlines a summary of studies in the defect prediction field.

In terms of software components, Perl et al. [21] presented a software defect prediction model using *'commit'* as the component in the version control system (VCS), such as Git, Mercurial, CVS, or Subversion, and so on. It combined code metrics analysis with metadata gathered from code repositories to extract defect features from commits. The authors then used SVM to build the prediction model. The significant advantage of this model was the achievement of incremental detection, an efficient method that only required the newest software code. Furthermore, it showed high efficiency compared with models based on file components. However, this method can only be applied to softwares that use version control systems. Most studies used files as predictive components in their prediction models.

In regard to feature extraction, Scandariato et al. [23] proposed a novel feature extraction method that used tokens as defective features based on a text mining technique. They employed bag-of-words representations, in which a source file was treated as a series of terms with associated frequencies. The terms were extracted as token vectors which were used as input of the classifier. Ultimately, they employed RF and NB algorithms as classifiers and applied them to 20 Android projects to build a model of which the accuracy reached 80%. Meanwhile, Wang et al. [25] presented a model that used Deep Belief Network (DBN) to automatically generate semantic features of the defective files. The model encoded code sequences based on the software AST and then inputted the code sequences into DBN to generate semantic features. The experiment result showed that the semantic features significantly improved both WPDP and CPDP mode compared to traditional features.

In view of ML algorithms, Malhotra [17] proposed an effective empirical framework for defect prediction using 18 ML algorithms, such as NB, LogitBoost, and multilayer perceptron. The results indicated that NB, LogitBoost, and multilayer perceptron have the best performance in defect prediction.

In sum, the models discussed above are innovative for feature extraction and ML algorithms, and most of them have a satisfactory performance. Nevertheless, these models cannot be easily applied to detect defects in apks. We thus propose a defect prediction model oriented to Android apks. The method extracts features from smali files using the smali2vec method and constructs comprehensive features combined token and semantic features. It employs DNN to train the model for more efficient learning and higher accuracy.

[2] OWASP, https://www.owasp.org/index.php/Category:Vulnerability.

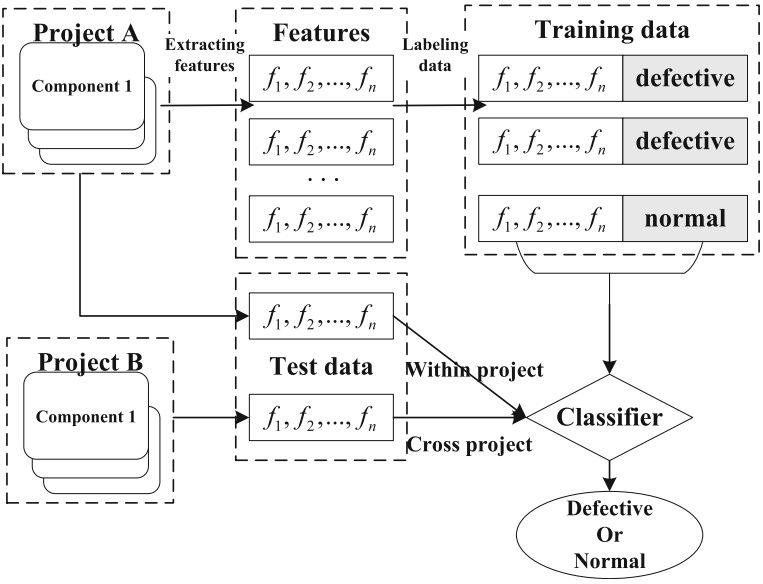

Fig. 1 General process of software defect prediction

Table 1 Summary of studies of software defect prediction

Study	Feature	Classifier
Perl et al. [21]	Code metrics	SVM
Scandariato et al. [23]	Token features	NB and RF
Wang et al. [25]	Semantic features	DBN and NB and LR
Malhotra [17]	Code metrics	18 ML algorithms
Zhang et al. [29]	Code metrics	RF and NB and LR and DT

2.2 Smali Files

Smali files are decompiled files derived from dex files of apks and have an one-to-one match with the source code java files. It is obvious that each source code java file has a counterpart in the list of smali files. In other words, when we find the defects of an a.smali file, we can conclude that defects exist in the corresponding source code a.java file. Furthermore, the identification, such as *'.line'* in the smali files, can directly indicate the source code position, which is helpful for identifying the defect location. As shown in Fig. 2, the file a.java corresponds with a.smali after decompiling. Taking a.smali as an example, the identification *'.line 15'* indicates the following codes correspond to the source codes in the fifteenth line in a.java.

Software source code has token features based on string text, as discussed above. These features are eligible for use in code clone detection and defect prediction tasks, as indicated by previous research [2, 23]. Moreover, these relevant tasks can employ the semantic features that represent the semantic characteristics of software based on the AST, PDG, and CFG [5, 30, 31]. Similar to source code, smali files also have token features based on the frequency of instructions and semantic features based on the sequence of instructions

```
12  protected void onCreate(Bundle savedInstanceState) {
13      super.onCreate(savedInstanceState);
14      setContentView(R.layout.activity_main);
15      myTextView = (TextView)findViewById(R.id.Text_view);
16
17      if(random() >= 5)
18        myTextView.setText("Hi~~");
19  }
```

a.java

Part of a.smali

```
# virtual methods
.method protected onCreate(Landroid/os/Bundle;)V
   .locals 2
   .param p1, "savedInstanceState"    # Landroid/os/Bundle;
   .prologue
   .line 13
   invoke-super {p0, p1}, Landroid/support/v7/app/AppCompatActivity;-
>onCreate(Landroid/os/Bundle;)V
   .line 14
   const v0, 0x7f040019
   invoke-virtual {p0, v0}, Lcom/example/maomao/helloworldapp/
MainActivity;->setContentView(I)V
   .line 15
   const v0, 0x7f0c0050
   invoke-virtual {p0, v0}, Lcom/example/maomao/helloworldapp/
MainActivity;->findViewById(I)Landroid/view/View;
   move-result-object v0
   check-cast v0, Landroid/widget/TextView;
   iput-object v0, p0, Lcom/example/maomao/helloworldapp/MainActivity;-
>myTextView:Landroid/widget/TextView;
...
.end method
```

Fig. 2 A.java corresponds to a.smali

[18, 19]. For token features, the smali file can be viewed as a bag of dalvik instructions. The frequency of dalvik instructions can show the token properties of smali files. Meanwhile, semantic features can indirectly reflect the semantic characteristics of smali files [10]. It has been proved that software semantic features is useful for bug detection and defect prediction [25]. We present smali2vec to generate the smali vector that combines token features and semantic features to represent defects in smali files. Details are discussed in Sect. 3.4.

2.3 Deep Neural Network

Deep networks refer to wide classes of machine learning techniques that exploit many layers of non-linear information processing for supervised or unsupervised feature extraction and transformation, and for pattern analysis and classification [4]. It has made many breakthroughs in many filed of artificial intelligence [9, 27]. More and More researchers begin to apply deep networks to the field of software analysis for malware detection or defect prediction [3, 10, 19, 28]. Deep network model can be built by different architectures for different application scenarios [11, 24], e.g., deep neural network (DNN), convolutional neural network (CNN), recursive neural network (RNN), and deep belief network (DBN).

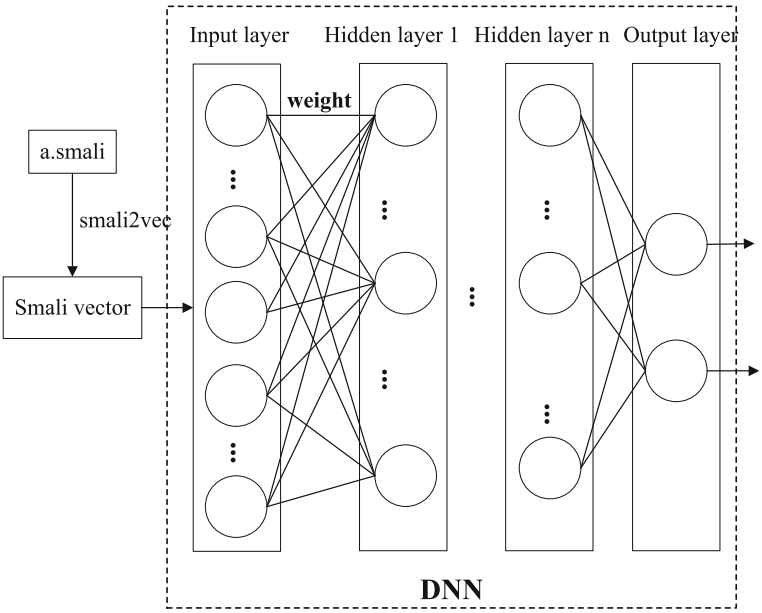

Fig. 3 DNN architecture with input instance of a.smali

Depending on the intention of usage, deep networks architectures can be categorized into two major classes:

1. Deep networks for unsupervised or generative learning, which are intended to capture feature or representation of the observed data in an unsupervised manner.
2. Deep networks for supervised learning, which are intended to predict the class labels of the observed data for pattern analysis and classification.

DNN, CNN, RNN are often used to build the supervised-learning models while DBN, regularized autoencoder are used to build the unsupervised-learning models [4].In this paper, we choose DNN to build a supervised-learning model for the defect prediction task. A DNN [11] is a conventional multi-layer perceptron with many hidden layers. As Fig. 3 shows, DNN has one input layer, one output layer and several layers of hidden layers. Each hidden layer has several neurons. After setting the number of neurons in each hidden layer and the number of hidden layers, DNN adjusts the weights between the neurons in different layers in the iteration process until it is well-tuned. We take the smali vector as the input of the input layer and train DNN to identify the defective smali files by adjusting parameters, including the number of hidden layers, the number of neurons in each layer, and the number of iterations. Details are described in Sect. 4.1.

3 Methodology

Figure 4 illustrates the overall architecture of our method. First, we collect apks and label defective smali files in them for training the model. To identify the defective smali files, we use Checkmarx, one of the most famous static source code analysis tools worldwide, to scan the corresponding source code of the smali files. Then, we present smali2vec to

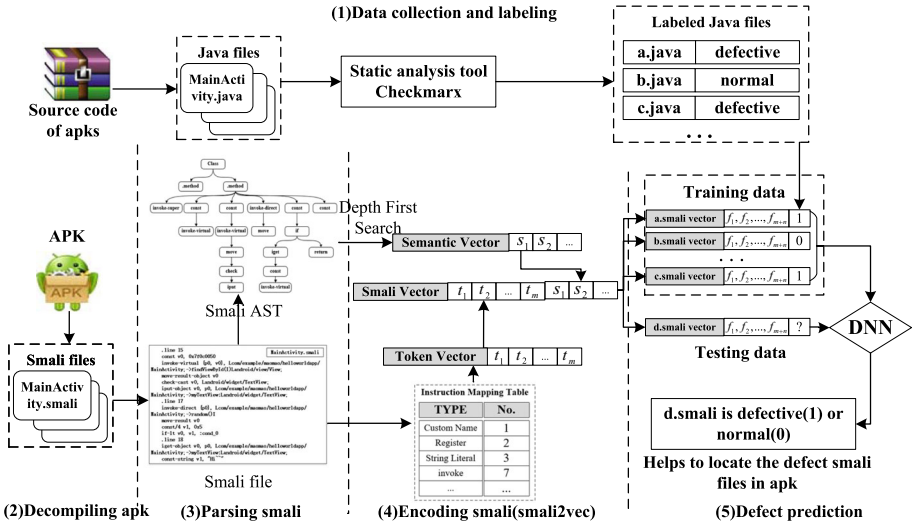

Fig. 4 Overview of the defect prediction in apks using DNN

generate the smali vector, which captures the characteristics of defects in smali files. The smali vector combines token features and semantic features of the smali file. We classify the instructions into eight categories and build the dalvik instruction mapping table. Token features are extracted from the decompiled smali files and are mapped to a token vector by recording the frequency of the instructions in the mapping table. Meanwhile, we use the depth-first search to traverse the AST of the smali files to generate semantic vectors. The two vectors merge to form the smali vector which is the input of the classifier. Finally, we use DNN to build our defect prediction model from the training data, and we predict defects from the test data or new apks.

In this paper, we design an accurate defect prediction model for Android apks. We first propose ***smali2vec*** to generate features that capture the characteristics of defective samli files in apks. Smali2vec generates the comprehensive features which capture both token and semantic features of defects in smali files of the apks. We classify the instructions into eight categories and build the dalvik instruction mapping table for the generation of token vectors and semantic vectors.

Our method consists of the following five steps: (1) collecting apks and labeling the defective smali files for training the model; (2) decompiling apk to smali files; (3) parsing smali to extract the smali vector; (4) encoding the smali file to the smali vector for the DNN input; and (5) training and building the defect prediction model using DNN.

3.1 Data Collection and Labeling

3.1.1 Android Applications Selection

We collect 92,220 smali files from 10 Android projects which are selected by a criteria. Both apks and their corresponding source code are needed because the source code is used to label the defects for training the model. Note that the source code is only needed in the training phase and we could handle apks without the source code after training the model.

Since both apks and the source code are required, we select the Android projects from GitHub.[3] Details of our selection criteria are as follows.

1. *The number of version is greater than 20.* Since different versions of a selected project are needed for the WPDP mode, the number of releases of a project becomes a key value of our selection criteria. In addition, the selected releases of a project require a certain interval between each other to avoid code duplication. We select five different versions for each project, and the interval between every two versions is set to at least four. Thus, the number of releases of the selected project should be greater than 20.
2. *Package size is greater than 500 KB.* To prevent low generalization ability of the prediction model caused by insufficient source code, the package size of the selected project should be greater than a certain number. All of our selected projects are greater than 500 KB.
3. *A high value of commit and contributor.* Commit means pushing changes to the remote package repository, while the contributor is the participant of the project. These two indicators represent the project activity. The greater is the value, the higher priority is the project. To build a common defect prediction model, we choose a project with relatively high commit and contributor values.

Previous Android app defect prediction models [17, 23] choose various projects of Android operating systems (e.g., Contact, Multimedia Messaging Service, Bluetooth, Email) as their dataset. However, although the system apps are open source online, certain system apps, such as Contact, are designed to be built as part of a firmware image, not as standalone apps. They are private/internal packages and have many classes and variables that are not presented in the Android software development kit (SDK). To build the system apps, it is necessary to download the entire system source code according to XDA forum,[4] the world's most popular Android developer forum. Furthermore, our model is designed for third-party apps, not for system apps. Thus, we choose apps on the GitHub as our dataset.

Table 2 shows the results of our selection. Fifty projects within ten apps (five versions in each Android app) are chosen from over thousands of Android repositories presented in GitHub. They meet all of our selection criteria above.

3.1.2 Defective Smali File Labeling

In this section, the aim is to locate the defective codes in every smali file and label the defective files for training the model. A smali file containing one or more defect is labeled as defective; otherwise, it is a normal file. The defective file is labeled by 1, whereas 0 represents normal. All of the smali files from the selected projects must be labeled for the model input.

However, the fifty projects containing more than 90,000 smali files are too much for us to manually perform the identification work. Furthermore, few Android defects are reported by the Android Vulnerability Database (AVD)[5] or Common Vulnerabilities and Exposures (CVE).[6] To date, there are only 19 Android apk defects in AVD. An automated

[3] Github, https://github.com.

[4] Xda Forums, Compiling AOSP Standalone apps: https://forum.xda-developers.com/showthread.php?t=1800090.

[5] AVD, Android Vulnerabilities Database, http://android.scap.org.cn/.

[6] CVE, Common Vulnerabilities and Exposures, http://cve.mitre.org/.

Table 2 Android projects selection result

Project	Category	Size (KB)	Releases	Commits	Contributors
D1: AnkiDroid	Education	8321	575	8565	102
D2: BankDroid	Money	1476	55	1310	39
D3: BoardGameGeek	Reading	697	44	3242	4
D4: Chess	Games	1318	24	340	6
D5: ConnectBot	Internet	1262	281	1475	32
D6: Andlytics	Tools	553	25	1478	27
D7: FBreader	Reading	2262	404	9016	35
D8: K9Mail	Internet	3450	341	6654	149
D9: Wikipedia	Tools	3948	119	4307	43
D10: Yaaic	Internet	511	23	1063	19

static code analysis tool with a rich set of defect rules is suitable for our identification work. Since the smali file has an one-to-one match with the java class file, we identify defects in the source code of apks to label the defective smali file.

We use Checkmarx CxEnterprise, one of the most renowned static source code analysis tools worldwide, to locate and label the defective smali files. Our 7.1.6 HF6 version of Checkmarx is deployed on a 2.60 GHz Intel(R) Xeon(R) E5-2650 v2 CPU and a 32 GB RAM machine with the Android rules set. Checkmarx generates a detailed defect report for every input source code package. The report contains the type of defect, such as *'Information Leak'* or *'Integer Overflow'*, as well as the risk level, which is rated as high, middle, or low. Figure 5 shows a part of the check result of ***DccFileTransfer.java*** in selected app ***Yaaic version 0.1***. It is evident that there is an unreleased resource defect in line 129,

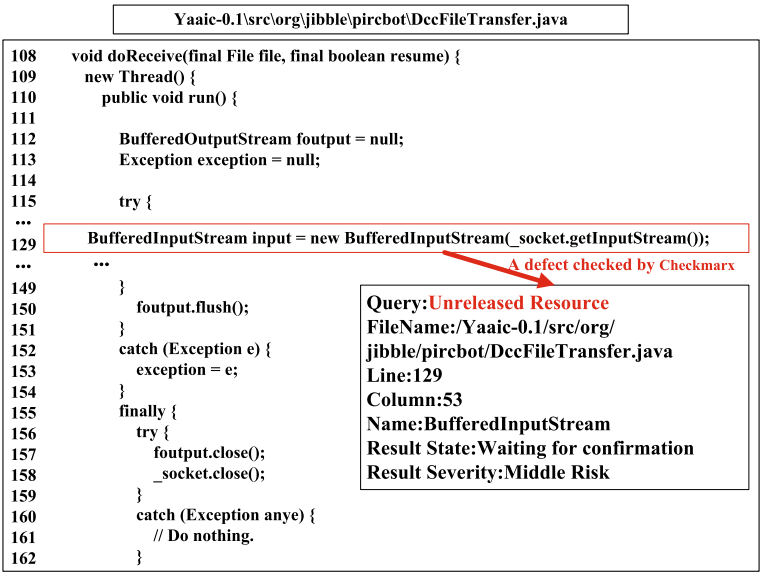

Fig. 5 Defect example checked by Checkmarx

column 53, with a medium risk. The program fails to release the *'BufferedInputStream'* input that could lead to system resource exhaustion. The result means that ***DccFileTransfer.smali*** is a defective file.

Every package of our ten selected projects was scanned by Checkmarx to label all Java class files. Table 7 in appendix shows the source code size, apk size, line of code (LOC), number of smali files, number of defective smali files, and percentage of defective smali files in each release of our selected apks. All datasets are available on the GitHub[7] for the research community to build a benchmark dataset for Android binary executable defect prediction.

3.2 Decompile Apk

The model unpack the fifty selected apk files to obtain the dex files and use the baksmali[8] tool to convert the dex files to smali files.

3.3 Parse Smali

The model parse the smali files to record the frequency of dalvik instructions for extracting token features. Details are shown in Sect. 3.4.1. To analyze the semantic features of smali files, the model use the sequence of the AST of smali files to build the semantic vector (details are given in Sect. 3.4.2).

3.4 Smali2vec

We propose a comprehensive feature generation method called smali2vec to generate the smali vector for smali file defect prediction. Smali code consists of instructions which include dalvik opcodes and some descriptors such as register, string and number. However, there are 245 types of dalvik opcodes in total according to the Android Open Source Project.[9] Using all the opcodes (called full-opcode) to extract defective features not only consumes a large amount of computational cost, but also pulls down the instructions ability of representing defect characteristics, thereby reducing model accuracy (described in Sect. 5.2) [14]. Thus we classify the instructions according to the official category of Android[10] and assign a sequence number for each instruction to build a mapping table for the generation of token vectors and semantic vectors.

As shown in Table 3, We classify instructions into eight categories:

1. *Custom name* Function or variable names which are defined by developers.
2. *Register* Registers are widely used in dalvik instructions because the dalvik machine is register-based.
3. *String literal* Constant strings which are used in the instructions.
4. *Class descriptor* Java class names in the instructions.
5. *Statement* Smali code expressions and statements.

[7] F. Dong, S.D. Zhang, S.H. Wang, DNN-based software defect prediction experimental data and code, https://github.com/breezedong/DNN-based-software-defect-prediction.

[8] JesusFreke, smali/baksmali, https://github.com/JesusFreke/smali/wiki.

[9] Dalvik opcodes: http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html.

[10] Android Open Source Project, Dalvik bytecode: https://source.android.com/devices/tech/dalvik/dalvikbytecode.

Table 3 Dalvik instruction mapping table

No.	Category	Detail
1	Custom name	Function or variable names which are defined by developers
2	Register	Android registers, such as v0, v1
3	String literal	Constant strings used in dalvik instructions
4	Class descriptor	Java class descriptors
5	Statement	Smali code expressions and statements
6	Number literal	Constant numbers used in the instructions
7–23	Function instruction	"invoke", "return", "new", "move", "const", "monitor", "get", "put", "cmp", "if", "goto", "switch", "check", "throw", "instance-of", "array-length", "fill-array-data"
24–37	Math instruction	"neg-", "not-", "-to-", "add-", "sub-", "mul-", "div-", "rem-", "and-", "xor-", "or-", "shl-", "ushr-", "shr-"

6. *Number Literal* Constant numbers which are used in the instructions
7. *Function instruction* Function instruction is instruction that performs functional operations, such as "move" instructions which move the source data to the destination data. Smali2vec finds 17 types of function instructions, including comparison instructions, and encodes them from 7 to 23 in turn. Note that most function instructions have a number of subclasses, we classify subclasses as one type. For example, smali2vec identifies "move-object" and "move-result" as "move". The same applies to Mathematical instruction.
8. *Mathematical instruction* Mathematical instruction is used for data computing. It includes three kinds of computing instructions, calculation instructions, logic instructions and displacement instructions. Calculation instructions, such as "add-", are used for adding, multiplication and division operations. Logic instructions, such as "and-", are used for nand operation which displacement instructions are used for shift operation.

3.4.1 Token Vector

In this section, we consider the extraction of the smali token features. Smali files can be viewed as a bag of smali instructions and the frequency of the instructions can capture the token features of the smali code in class level [23]. Because token features do not require consideration of the order of the instructions, we use the bag-of-words model, which was initially designed for analysis of text documents [23], to build the token vector. The frequency of all the 37 instructions in Table 3 is recorded in turn to generate token vector T $T(t_1, t_2, \ldots, t_{37})$. For ***a.smali*** in Fig. 2, token vector T is [1, 12, 0, 5, 8, 2, 3, 0, 0, 1, 2, 0,..., 0, 0, 0, 0]. The value of t_7 is equivalent to three, meaning that instruction ***"invoke"*** appears three times in ***a.smali***.

3.4.2 Semantic Vector

Software has well-defined syntax, which can be presented by abstract syntax trees and have been successfully used to capture the semantic features [25]. Moreover, the AST of smali

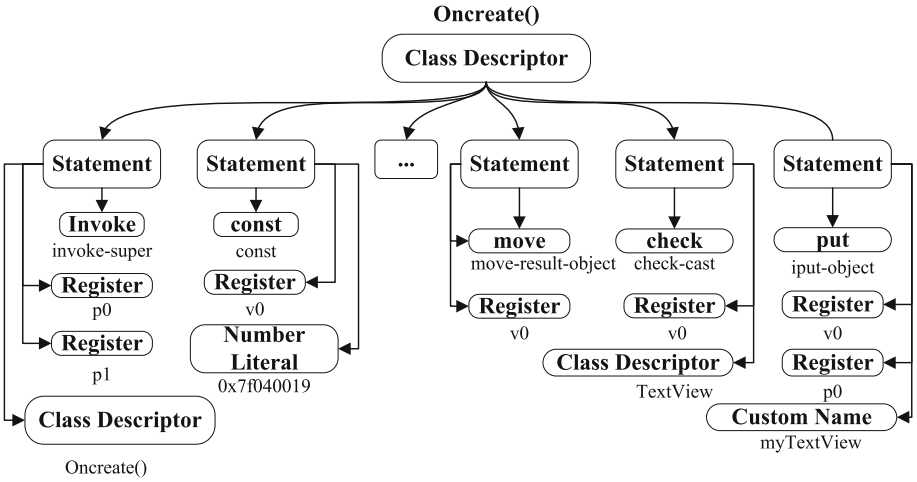

Fig. 6 AST of a.smali

files can reflect the semantic features in the dalvik code level. Thus, we build semantic vector $S(s_1, s_2, \ldots)$ by recording the sequence of the AST of smali files. First, we use antlr[11] (a powerful parser generator) to extract AST from the smali files. Then we apply depth-first search (DFS) traversal strategy for the AST to generate the AST sequence. At last, the AST sequence is encoded to an integer sequence according to the davik instruction mapping table in Table 3. For example, Fig. 6 shows the AST of a.smali in Fig. 2, the AST sequence generated by DFS is [***Class Descriptor, Statement, invoke, Register, Register, Class Descriptor,..., Statement, put, Register, Register, Custom Name***]. It can be mapped into semantic vector [4, 5, 7, 2, 2, 4, ..., 5, 14, 2, 2, 1].

3.4.3 Smali Vector

After generating the token vector and the semantic vector according to the frequencies of the dalvik instructions and the order of the AST of smali files, we splice these two vectors to form the smali vector. Since DNN takes the same lengths of the smali vectors as input, we append 0 to the smali vectors to make all the lengths consistent and equal to the length of the longest vector. As an example, we take token vector T in Sect. 3.4.1 and semantic vector S in Sect. 3.4.2. These two vectors are combined into the comprehensive smali vector [1, 12, 0, 5, 8, 2, 3, 0, 0, 1, 2, 0,..., 0, 0, 0, 0, 4, 5, 7, 2, 2, 4,..., 5, 14, 2, 2, 1, 0, 0, ...]. Adding zero to the tail of the vector makes it acceptable by DNN.

3.5 Prediction

After obtaining the smali vectors of all smali files, we use the training data to build our defect prediction model and use the test data to evaluate the performance.

[11] http://www.antlr.org/.

4 Experimental Setup

We developed smali2vec with Java. The DNN was implemented using tensorflow,[12] an open-source deep learning framework developed by Google. Our experimental code is available on GitHub [6]. All experiments were performed on a machine with an Intel(R) Xeon(R) CPU E5-2620 v3 running at 2.40 GHZ with 64 GB of RAM and 2 TB of ROM. We choose ROC-AUC as our performance measure, as Lessmann et al. [12] and Ghotra et al. [7] suggested.

4.1 Parameter Setting for Training DNN

To train an effective DNN for predicting defects in apks, our model needs to tune three parameters including the number of hidden layers, the number of neurons in each hidden layer and the number of iterations [28]. We tune the three parameters by conducting experiments with different values of these parameters on our training data, which is described in Sect. 5.1. We use the AUC value and the error rate to evaluate the performance and find the best configuration of the parameters.

4.1.1 Setting the Number of Hidden Layers and the Number of Neurons in Each Layer

We together adjust the number of hidden layers and the number of neurons in each layer because these two parameters always interact with each other. We conduct an experiment using ten values for the number of hidden layers, including 3, 5, 10, 15, 20, 25, 30, 50, 100, and 200. We set the number of neurons to be same in each layer to simplify our model. The number set contains 8 values, including 20, 50, 100, 200, 300, 500, 800, and 1000. When we evaluate these two parameters, we set the number of iterations to 10,000. Furthermore, a more accurate depiction is presented with the results in Sect. 5.1.

4.1.2 Setting the Number of Iterations

The number of iterations is another important parameter for building an effective DNN. DNN adjusts the weights between different neurons to narrow down the error rate in each iteration. A bigger number of iterations means a lower error rate, but it cost more time. We need to find the trade-off between the number of iterations and the cost of time.

To achieve this balance, we conduct an experiment on ten projects in WPDP with ten values of iterations including 1000, 2000, 3000, 5000, 10,000, 15,000, and 20,000. Meanwhile, we monitor the time cost and use the average error rate to evaluate this parameter.

4.2 Setup for Traditional ML Algorithms

In our study, we introduce four ML algorithms (SVM, NB, C4.5, and LR), which have the best performance in the field of software defect prediction [12, 17], to compare with DNN. We use the default configuration of the selected traditional ML algorithms to simplify the experiments.

[12] Google, TensorFlow Wide Deep Learning Tutorial, https://www.tensorflow.org.

4.3 Validation Method

4.3.1 Within-Project Defect Prediction Using Fivefold Cross Validation

There are ten datasets marked as $[D_1, D_2, \ldots, D_{10})]$, each of which had five samples and a large number of smali files. We conduct 10 sets of WPDP experiments, each of which contains training and testing smali files from the same dataset. To avoid over-fitting, we use fivefold cross validation [23] on each dataset. For example, 15,360 smali files in dataset D_1 are divided into five equal parts. We arbitrarily choose four for training, while the rest one is used for evaluation. This process is repeated five times, and the result of each fold is used to compute an average performance of our model on dataset D_1.

4.3.2 Cross-Project Defect Prediction

For CPDP, we simply take one dataset as the training data and the first version of another dataset is used for testing. For example, we use all smali files in data set D_1 for training while one of the five samples in set D_2 is taken as the testing data. This process is iterated five times. We then compute the average AUC of these five results and verifiy the performance of our model in CPDP.

5 Result

In this section, we present our research questions, along with the approach and findings.

5.1 RQ1: Is Our Defect Prediction Model Capable of Predicting Defects in apks and What is the Best Configuration of the DNN Classifier?

Approach We conduct experiments on ten datasets using the WPDP validation to find out that the capacity of predicting defects in apks of our model and the best configuration of it. The experimental method of WPDP is described in Sect. 4.3.1 and the setting of the three parameters of DNN is described in Sect. 4.1. Ten WPDP validation experiments are done on ten datasets with the same sets of parameters of DNN and the average AUC is computed to evaluate the performance of the model.

Findings Figure 7 shows the performance of our model with different number of neurons and number of hidden layers. In this experiment, we try to figure out the best number of neurons in each layer and the best number of hidden layers for DNN by AUC. As we can see, the average AUC of the green curve which has 300 neurons in each layer outperforms other curves in almost all the number of layers. Then we choose the number of neurons in each layer as 300. We also find that the AUC of the 300 neurons curve reaches the highest value of 85.98% at the 10 hidden layers, thus the best number of hidden layers is set to 10. Figure 8 shows the relation between the error rate and the time cost as the number of iterations increasing. As we can see, the time cost increases rapidly and the error rate decreases slowly when the number of iterations exceeds 10,000. The result shows that the defect prediction model is capable of predicting defects in apks with a high AUC of 85.98%. We set the number of hidden layers as 10, the number of neurons as 300 and the number of iterations as 10,000 for the best configuration.

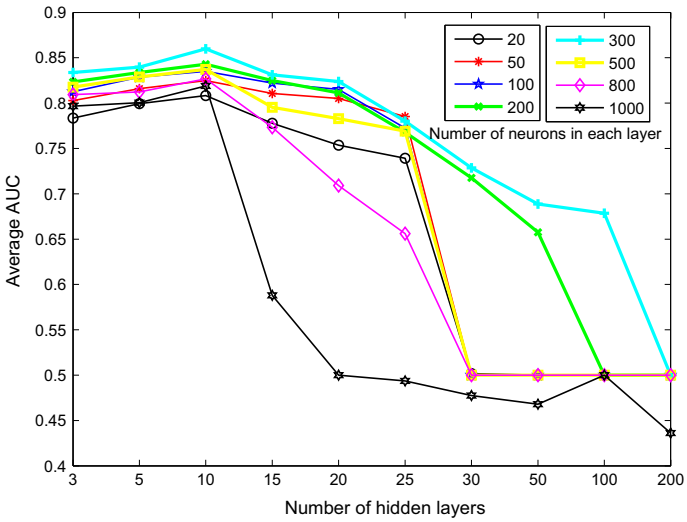

Fig. 7 DNN performance with different parameters

5.2 RQ2: Does the Comprehensive Features Based on the Mapping Table Outperform the Full-Opcode Features?

Approach To answer this question, we use all the 245 opcodes to encode the token features and semantic features of smali files to generate defective features (called full-opcode features). Then we compare the accuracy of the models generated by the comprehensive features based on the mapping table and the full-opcode features. Experiments are conducted in WPDP mode.

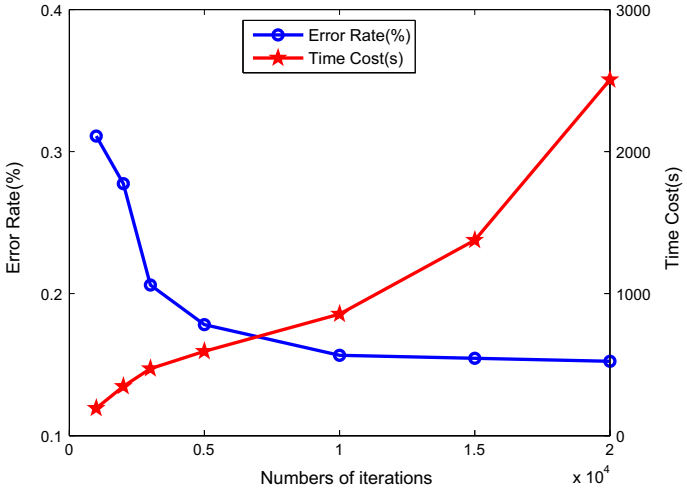

Fig. 8 Error rate and time cost for different iterations

Table 4 Comparison between comprehensive features and full-opcode features

Project name	DNN (smali2vec)	DNN (full-opcode)
AnkiDroid	0.7914	0.7756
BankDroid	0.7967	0.7967
BoardGameGeek	0.8887	0.8045
Chess	0.8481	0.7802
ConnectBot	0.9516	0.8835
Andlytics	0.8340	0.7756
FBreader	0.8932	0.8039
K9Mail	0.7655	0.6256
Wikipedia	0.8922	0.8119
Yaaic	0.9371	0.9184
Average AUC	0.8598	0.7976

Findings As shown in Table 4, the AUC of the comprehensive features is 6% higher than that of the full-opcode features. Besides, the full-opcode features consume more time and space, proving that the comprehensive features are more suitable for the model.

5.3 RQ3: Does the DNN Algorithm Outperform the Traditional ML algorithms in Within-Project Defect Prediction Mode?

Approach We select four traditional ML algorithms (SVM, NB, C4.5, and LR) as the classifiers to compare with DNN using WPDP validation. We set the parameter of DNN with the result of RQ1 and the parameters of traditional ML algorithms are described in Sect. 4.2. Each algorithm conducts ten sets of WPDP experiments on the selected datasets. We compute the average AUC to compare the performance of the classifiers.

Findings Table 5 shows the results of experiments using different algorithms in WPDP. From the table, it is apparent that the average AUC of the models is more than 70%, which highlights the capability of our comprehensive features extracted by smali2vec. It is also clearly evident that DNN significantly outperforms the traditional ML algorithms. The total average AUC value of DNN reaches 85.98% (bold in Table 5), which is approximately 8% higher than the average AUC of NB (77.93% which is bold in Table 5).

5.4 RQ4: Does the DNN Algorithm Outperform the Traditional ML Algorithms in Cross-Project Defect Prediction Mode?

Approach We repeat the experiments in RQ2 using CPDP validation. We take one dataset as training data (called source project) and one version of the five samples in another dataset as testing data (called target project). For example, we use K9Mail to predict defects in Yaaic. All smali files in source prject K9Mail are taken as training data, and one version of Yaaic are taken as testing data. The average AUC of the five versions of BankDroid is computed to evaluate the performance of each set experiment. There will be as many as 90 sets of CPDP experiments if every dataset is used to predict another one. So we simply choose ten sets CPDP experiments to find out the performance of the different algorithms. In the first five sets experiments, the source project and the destination project belong to the same category while they belong to different categories in the latter five sets.

Table 5 Comparison between DNN and traditional ML algorithms in WPDP

Project name	DNN	SVM	NB	C4.5	LR
AnkiDroid	0.7914	0.8056	0.9260	0.8294	0.8184
BankDroid	0.7967	0.6873	0.5970	0.7033	0.5429
BoardGameGeek	0.8887	0.7271	0.7341	0.7932	0.6793
Chess	0.8887	0.7271	0.7341	0.7932	0.6793
ConnectBot	0.9516	0.6465	0.6973	0.6648	0.7287
Andlytics	0.8340	0.7783	0.9427	0.8138	0.8173
FBreader	0.8932	0.8048	0.9084	0.8260	0.7034
K9Mail	0.7655	0.5391	0.5915	0.5473	0.5697
Wikipedia	0.7655	0.5391	0.5915	0.5473	0.5697
Yaaic	0.9371	0.7226	0.7706	0.7659	0.6966
Average AUC	**0.8598**	0.7279	**0.7793**	0.7490	0.7139

Table 6 Comparison between DNN and traditional ML algorithms in CPDP

Source	Target	DNN	SVM	NB	C4.5	LR
ConnectBot	Yaaic	0.7850	0.7301	0.6987	0.7850	0.7458
FBreader	BoardGameGeek	0.8058	0.6688	0.7575	0.7172	0.7219
K9Mail	Yaaic	**0.8318**	0.6950	0.6950	0.6178	0.6809
Yaaic	K9Mail	0.8020	0.6403	0.8401	0.8235	0.6821
Andlytics	Wikipedia	0.7963	0.6848	0.8282	0.8282	0.8361
BankDroid	Chess	0.6358	0.5913	0.5404	0.5659	0.6612
Chess	Andlytics	0.5950	0.5772	0.6177	0.5474	0.6248
Yaaic	Andlytics	0.5704	0.5248	0.5362	0.4962	0.5476
BoardGameGeek	BankDroid	0.6415	0.5090	0.5361	0.5144	0.4765
Chess	Andlytics	0.5368	0.5314	0.5563	0.4563	0.4563
Average AUC		**0.7**	0.6153	0.6606	0.6352	0.6433

Findings Table 6 shows the results of the five classifiers in cross-project defect prediction. As in most studies [17, 25, 29], the average performance of CPDP is much lower than that of WPDP. CPDP uses the defect knowledge learned from the source project to predict defects of the target project. The performance of CPDP is worse than WPDP in most cases due to the low correlation between the different projects. When the source project and target project are from different categories, such as Chess and Andlytics (from Games and Tools categories respectively), the AUC is 53.68%, which is far away from perfect. But the performance is close to that of the WPDP experiments when the two projects are from the same category (such as K9Mail and Yaaic with the AUC of 83.18%, which are both from the Internet category). So we conclude that the accuracy of CPDP is related to the category of the projects. Besides, the average AUC indicates that DNN (70% which is bold in Table 6) outperforms the traditional ML algorithms in CPDP mode.

5.5 RQ5: What is the Time and Space Cost of the DNN Algorithm Versus Traditional ML Algorithms?

Approach We track the time and memory space costs of the training and testing processes on the ten sets of each classifier in the RQ3 WPDP experiments. For the other processes, such as data input and feature generation, the procedures are all the same before applying each classifier. Thus, we do not analyze the related time and space costs.

Findings The results of average time and memory costs of the five different models on ten projects are shown in Fig. 9. Obviously, DNN consumes 487 MB of memory and 832.5 s of time, which is the most resources-intensive. Although the time and space cost of DNN is much higher than others, the cost of resource consumption is acceptable relative to the good performance of DNN.

6 Discussion

In this section, we examine possible limitations of our model and discuss potential future improvements.

6.1 Data Collection and Labeling

We select 50 Android apks with source code as our datasets due to lack of benchmark of Android defect dataset. There are two main weaknesses which may affect the accuracy of the model. First, the smali files in the selected apks cannot cover all types of defects. It may make the model unable to predict the types of defects which are not covered. The experimental results show that the current selection is acceptable and we will adjust the selection to improve the defects coverage of the datasets. Another weakness is the accuracy of Checkmarx. Similar to other static source code analysis tools, Checkmarx has a certain error rate which may cause the error of the model. But the experimental results show that the cost of error rate is acceptable and much lower than that of manually validating all smali files of the selected apks. Manual validation of Checkmarx reports requires long-term

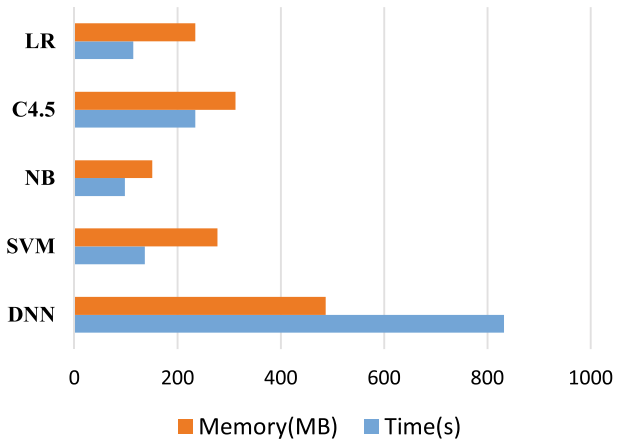

Fig. 9 The time and space cost of different algorithms

work. So all datasets are available on the GitHub [6] for the research community to improve and build a benchmark dataset for apk defect prediction.

6.2 Applying the Model to Other Platforms

Our model predicts defective smali files in apks based on the fact that apks can be decompiled into a number of disassembly files. For other platforms, such as iOS, our model cannot be applied for iOS binary executables (called ipa) defect prediction because there is only one disassembly file decompiled from ipa. Thus we can only use ipa files as components to build the model. However, it does not make sense to predict whether an ipa has a defect, because the defect needs to be located. Although our model cannot be applied directly, we could divide the disassembly file into code segments and take one of the segments as predicting module to build the model in future.

7 Conclusions

In this study, we design a practical defect prediction model for apks. The model leverages a new approach, smali2vec, to generate the comprehensive features which capture both token and semantic features of defects in smali files of the apks. Furthermore, we employ DNN instead of the traditional shallow ML algorithms to train and build the model. Moreover, to avoid over-fitting, we use fivefold cross validation to divide the training and test data. After the model is fully trained, it locates the defective smali files in the new input apks without the source code.

We apply the prediction model to a large dataset containing 92,220 smali files from 50 apks. The results show that the DNN-based model is capable of predicting defects in apks with a high AUC of 85.98% in WPDP mode and an AUC of 70% in CPDP mode. The DNN algorithm outperforms the traditional ML algorithms in both WPDP and CPDP mode. The proposed model has been used by our app defect prediction team in our practical work. It has helped predict many instances of app defects with only apks input.

For future work, we intend to expand our datasets to more Android projects so that the model can be generalized to more apks. Our datasets are available on GitHub for the research community to enable building a benchmark for defect prediction of apks. We additionally intend to transform the DNN-based model into a distributed version to reduce the related time and space costs.

Acknowledgements This research is supported by the National Natural Science Foundation of China Project (No. 61401038) and the 2016 Frontier and Key Technology Innovation Project of Guangdong Province Science and Technology Department (No. 2016B010110002). It is also supported by the National Key Research and Development Program (2016QY06X1205) and Technology Research and Development Program of Sichuan, China (17ZDYF2583).

Appendix: Results of Labelling of the Selected App Projects

See Table 7.

Table 7 Results of labelling of the selected application projects

Project	Version	Size (KB)	Smali	LOC	Defects	Defective (%)
AnkiDroid	2.1	7455	3154	77,624	558	17.69
	2.5	8000	3141	78,164	553	17.61
	2.5.4	8150	3070	74,209	556	18.11
	2.8.2	9434	2940	74,719	595	20.24
	2.9	9169	3055	75,284	596	19.51
BankDroid	1.9.5.4	835	1673	32,202	151	9.03
	1.9.6.4	917	1706	30,502	166	9.73
	1.9.7.4	2360	3265	20,527	197	6.03
	1.9.8.0	1610	3564	26,080	198	5.56
	1.9.9.5	1660	3667	23,336	192	5.24
BoardGame Geek	3	289	170	8838	39	22.94
	3.5	597	337	18,744	40	11.87
	3.6	714	449	23,390	170	37.86
	4	877	1072	61,953	286	26.68
	4.7	1010	1638	73,959	338	20.63
Chess	1.0.1	484	747	23,669	71	9.50
	1.1.0	485	746	24,083	57	7.64
	8.5.0	2831	1980	24,532	63	3.18
	8.6.1	3577	690	24,961	98	14.20
	8.6.7	2120	2924	25,387	72	2.46
ConnectBot	1.5.0	1827	733	90,626	160	21.83
	1.6.0	969	758	52,937	122	16.09
	1.7.0	1450	377	90,283	180	47.75
	1.8.4	1470	378	52,821	179	47.35
	1.8.6	1310	1460	23,445	55	3.77
Andlytics	2.1.0	518	747	1846	130	17.40
	2.2.3	495	846	1856	110	13.00
	2.3.0	553	919	2005	144	15.67
	2.4.0	568	923	2056	150	16.25
	2.5.0	630	960	2295	150	15.63
FBreader	2.3.3	598	1286	62,762	265	20.61
	2.4.4	858	2257	64,724	297	13.16
	2.4.7	1315	2225	63,649	320	14.38
	2.5.1	2270	2257	64,639	502	22.24
	2.5.9	2300	2262	66,784	519	22.94
K9Mail	5.206	2037	1909	86,782	208	10.90
	5.2	2198	1330	86,967	252	18.95
	5.1	2770	1907	87,349	224	11.75
	5.105	2730	3028	86,853	263	8.69
	5.11	6190	3444	14,703	381	11.06

Table 7 continued

Project	Version	Size (KB)	Smali	LOC	Defects	Defective (%)
Wikipedia	2.0.106	3220	2166	31,382	429	19.81
	2.1.131	2860	4078	42,019	562	13.78
	2.2.146	4350	5048	54,114	742	14.70
	2.3.152	4630	5372	60,261	912	16.98
	2.4.160	4680	5401	60,935	894	16.55
Yaaic	0.1	336	80	10,936	41	51.25
	0.4	408	105	14,111	49	46.67
	0.7	493	120	17,610	60	50.00
	0.9	569	130	19,781	68	52.31
	1.1	749	1030	19,289	73	7.09

References

1. Bengio, Y. (2009). Learning deep architectures for ai. *Foundations & Trends in Machine Learning*, *2*(1), 1–127.
2. Bishnu, P. S., & Bhattacherjee, V. (2012). Software fault prediction using quad tree-based k-means clustering algorithm. *IEEE Transactions on Knowledge and Data Engineering*, *24*(6), 1146–1150.
3. David, O. E., & Netanyahu, N. S. (2015). Deepsign: Deep learning for automatic malware signature generation and classification. In *International Joint Conference on Neural Networks* (pp. 1–8).
4. Deng, L., & Yu, D. (2014). Deep learning: methods and applications. *Foundations and Trends® in Signal Processing, 7*(3–4), 197–387.
5. Du, Y., Wang, X., & Wang, J. (2015). A static android malicious code detection method based on multisource fusion. *Security and Communication Networks*, *8*(17), 3238–3246.
6. Dong, S. Z., & Wang, S. (2017). Dnn-based software defect prediction experimental data and code, https://github.com/breezedong/DNN-based-software-defect-prediction. Accessed July 20, 2017.
7. Ghotra, B., Mcintosh, S., & Hassan, A. E. (2015). Revisiting the impact of classification techniques on the performance of defect prediction models. In *IEEE/ACM IEEE International Conference on Software Engineering* (pp. 789–800).
8. Hall, T., Beecham, S., Bowes, D., Gray, D., & Counsell, S. (2012). A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, *38*(6), 1276–1304.
9. Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A. R., Jaitly, N., et al. (2012). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, *29*(6), 82–97.
10. Jerome, Q., Allix, K., State, R., & Engel, T. (2014). Using opcode-sequences to detect malicious android applications. In *IEEE international conference on communications* (pp. 914–919).
11. Lecun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444.
12. Lessmann, S., Baesens, B., Mues, C., & Pietsch, S. (2008). Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, *34*(4), 485–496.
13. Ma, Z., Rana, P. K., Taghia, J., Flierl, M., & Leijon, A. (2014). Bayesian estimation of Dirichlet mixture model with variational inference. *Pattern Recognition, 47*(9), 3143–3157.
14. Ma, Z., Tan, Z. H., & Guo, J. (2016). Feature selection for neutral vector in eeg signal classification. *Neurocomputing*, *174*, 937–945.
15. Ma, Z., Teschendorff, A. E., Leijon, A., Qiao, Y., Zhang, H., & Guo, J. (2015). Variational bayesian matrix factorization for bounded support data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *37*(4), 876–889.
16. Ma, Z., Xie, J., Li, H., Sun, Q., Si, Z., Zhang, J., et al. (2017). The role of data analysis in the development of intelligent energy networks. *IEEE Network*, *31*(5), 88–95.
17. Malhotra, R. (2016). An empirical framework for defect prediction using machine learning techniques with Android software. *Applied Soft Computing, 49*, 1034–1050.

18. Mclaughlin, N., Rincon, J. M. D., Kang, B. J., Yerima, S., Miller, P., Sezer, S., et al. (2017). Deep android malware detection. In *ACM on conference on data and application security and privacy* (pp. 301–308).
19. Mou, L., Li, G., Jin, Z., Zhang, L., & Wang, T. (2014). Tbcnn: A tree-based convolutional neural network for programming language processing. Eprint Arxiv.
20. Nguyen, V. H., & Le, M. S. T. (2010). Predicting vulnerable software components with dependency graphs. In *International workshop on security measurements and metrics* (p. 3).
21. Perl, H., Dechand, S., Smith, M., Arp, D., Yamaguchi, F., Rieck, K., et al. (2015). Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *ACM Sigsac conference on computer and communications security* (pp. 426–437).
22. Prasad, M. C., Florence, L., & Arya, A. (2015). A study on software metrics based software defect prediction using data mining and machine learning techniques. *International Journal of Database Theory and Application, 8*(3), 179–190.
23. Scandariato, R., Walden, J., Hovsepyan, A., & Joosen, W. (2014). Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, *40*(10), 993–1006.
24. Schmidhuber, J. (2014). Deep learning in neural networks: An overview. *Neural Networks the Official Journal of the International Neural Network Society*, *61*, 85.
25. Wang, S., Liu, T., & Tan, L.: Automatically learning semantic features for defect prediction. In *IEEE/ACM international conference on software engineering* (pp. 297–308).
26. Xu, P., Yin, Q., Huang, Y., Song, Y. Z., Ma, Z., Wang, L., & Guo, J. (2017). Cross-modal Subspace Learning for Fine-grained Sketch-based Image Retrieval. arXiv preprint arXiv:1705.09888.
27. Xu, Y., Du, J., Dai, L. R., & Lee, C. H. (2013). An experimental study on speech enhancement based on deep neural networks. *IEEE Signal Processing Letters*, *21*(1), 65–68.
28. Yuan, Z., Lu, Y., Wang, Z., & Xue, Y. (2014). Droid-sec: Deep learning in android malware detection. *ACM Sigcomm Computer Communication Review*, *44*(4), 371–372.
29. Zhang, F., Zheng, Q., Zou, Y., & Hassan, A. E. (2016). Cross-project defect prediction using a connectivity-based unsupervised classifier. In *IEEE/ACM international conference on software engineering* (pp. 309–320).
30. Zhao, Z., Wang, J., & Bai, J. (2013). Malware detection method based on the control-flow construct feature of software. *Iet Information Security*, *8*(1), 18–24.
31. Zhao, Z., Wang, J., & Wang, C. (2013). An unknown malware detection scheme based on the features of graph. *Security and Communication Networks*, *6*(2), 239–246.

Feng Dong received the M.S. degree in communication engineering from Wuhan University of Technology, Wuhan, China, in 2012. He is currently pursuing the Ph.D. degree in information security at Beijing University of Posts and Telecommunication. His research interests are in the areas of software security, machine learning and data analysis.

Junfeng Wang received the M.S. degree in Computer Application Technology from Chongqing University of Posts and Telecommunications, Chongqing in 2001 and Ph.D. degree in Computer Science from University of Electronic Science and Technology of China, Chengdu in 2004. From July 2004 to August 2006, he held a postdoctoral position in Institute of Software, Chinese Academy of Sciences. Dr. Wang is with the School of Aeronautics and Astronautics and the College of Computer Science, Sichuan University as a professor. He is currently serving as an associate editor for IEEE Access, IEEE Internet of Things and Security and Communication Networks, etc. His recent research interests include network and information security, spatial information networks and data mining.

Qi Li received the Ph.D. degree in computer science and technology from Beijing University of Posts and Telecommunications, China, in 2010. She is currently an associate professor in the Information Security Center, State Key Laboratory of Networking and Switching Technology, School of Computer Science, Beijing University of Posts and Telecommunications, China. Her current research focuses on information systems and software security.

Guoai Xu received the Ph.D. degree in signal and information processing from Beijing University of Posts and Telecommunications, China, in 2002. He is currently an associate director in the National Engineering Laboratory of Security Technology for Mobile Internet, School of Computer Science, Beijing University of Posts and Telecommunication. His research interests are in the areas of software security and data analysis.

Shaodong Zhang received the B.S. degree in computer science and technology from Anhui Agricultural University, Anhui, China, in 2016. He is currently pursuing the M.S. degree in information security at Beijing University of Posts and Telecommunication. His research interests are in the areas of software security and data analysis.