CDGDroid: Android Malware Detection Based on Deep Learning using CFG and DFG

Zhiwu Xu[1,2], Kerong Ren[1], Shengchao Qin[3], and Florin Craciun[4]

[1] College of Computer Science and Software Engineering, Shenzhen University, China
[2] National Engineering Laboratory for Big Data System Computing Technology, Shenzhen University, China
[3] School of Computing, Media and the Arts, Teesside University, UK
[4] Faculty of Mathematics and Computer Science, Babes-Bolyai University, Romania
`xuzhiwu@szu.edu.cn, renkerong99@foxmail.com, shengchao.qin@gmail.com, craciunf@cs.ubbcluj.ro`

Abstract. Android malware has become a serious threat in our daily digital life, and thus there is a pressing need to effectively detect or defend against them. Recent techniques have relied on the extraction of lightweight syntactic features that are suitable for machine learning classification, but despite of their promising results, the features they extract are often too simple to characterise Android applications, and thus may be insufficient when used to detect Android malware. In this paper, we propose CDGDroid, an effective approach for Android malware detection based on deep learning. We use the semantics graph representations, that is, control flow graph, data flow graph, and their possible combinations, as the features to characterise Android applications. We encode the graphs into matrices, and use them to train the classification model via Convolutional Neural Network (CNN). We have conducted some experiments on Marvin, Drebin, VirusShare and ContagioDump datasets to evaluate our approach and have identified that the classification model taking the horizontal combination of CFG and DFG as features offers the best performance in terms of accuracy among all combinations. We have also conducted experiments to compare our approach against Yeganeh Safaei et al.'s approach, Allix et al.'s approach, Drebin and many antivirus tools gathered in VirusTotal, and the experimental results have confirmed that our classification model gives a better performance than the others.

1 Introduction

According to a report from IDC [1], Android is the most popular platform for mobile devices, with almost 85% of the market share in the first quarter of 2017. Unfortunately, the increasing adoption of Android comes with the growing prevalence of Android malware. A report from security firm G DATA [2] shows that a new instance of Android malware pops up nearly every 10 seconds. Consequently, Android malware has become a serious threat for our daily life, and thus there is a pressing need to effectively mitigate or defend against them.

To protect legitimate users from the threat, many approaches and tools to detect Android malware have been proposed over the past decade. These approaches can be summarised into two categories, namely, the approach based on program analysis techniques and the approach based on machine learning techniques. The first approach aims to identify the malicious code patterns in Android applications, through either static analysis [3–5] or dynamic analysis [6–8]. But the high overhead and the rapid evolution of Android malware make this approach no longer effective. Recently, various machine learning techniques like support vector machine, decision tree and deep learning have been proposed for detecting Android malware [9–14]. This approach constructs a learning-based classification model through a (big) dataset. The key of this approach is to seek out an appropriate feature set, such as permissions, APIs, and opcodes. However, despite of their promising results, the features that are considered by most existing work based on machine learning are often too simple to characterise Android applications (e.g., lack of either control flow information or data flow information) or non-robust (e.g., prone to suffering from the poisoning attack) [15], and thus may be insufficient to help detect Android malware.

In this paper, we propose CDGDroid, an effective approach to detecting Android malware based on deep learning. Different from most existing work based on machine learning, we use two classic semantic representations of programs in program analysis techniques, namely, control flow graphs and data flow graphs, as the features to characterise Android applications. Generally, graphs offer a natural way to model the sequence of activities that occur in a program. Hence they serve as amenable data-structures for detecting malware through identifying suspicious activity sequences. In particular, a control flow graph reflects what a program intends to behave (*e.g.*, opcodes) as well as how it behaves (*e.g.*, possible execution paths), such that malware behaviour patterns can be captured easily by this feature. For example, Geinimi samples share the similar control flow graphs. On the other hand, a data flow graph represents the data dependancies between a number of operations, and thus can help in detecting malware involving sensitive or network data, like HippoSMS and RogueSppush that send and block SMS message in the background.

Our approach consists of two phases: the first phase aims to learn a classification model from an existing dataset; and the second phase uses this model to detect new, unseen malicious and normal applications. In detail, we extract control flow graphs and data flow graphs in the instruction level from applications, which are collected through static analysis on the *smali* files (*i.e.*, Dalvik executions) in applications. Both intra-procedural analysis and inter-procedural analysis are considered for these two graphs. We then encode control flow graphs and data flow graphs into matrices, where only the opcodes are preserved. Meanwhile, their possible combination modes of control flow graph and data flow graph are considered as well: two graphs are combined either via the matrix addition (called the *vertical* mode) or via the matrix extension (called the *horizontal* mode). Finally, the encoded matrices are fed into the classification model for training or testing. We use a convolutional neural network (CNN for short), a new frontier in machine learning that has successfully been applied to analyse visual imagery (*i.e.*, matrix data), to build our model.

Several experiments have been conducted to evaluate our approach. We first conduct 10-fold cross validation experiments to see the effectiveness of CFG and DFG in malware detection. We have found that the classification model with the horizontal combination of CFG and DFG as features performs the best, with the *F1 score* (a measure of a test's accuracy, see Section 3.1) 98.722%. We also run our model on datasets consisting of new, unknown samples. The experimental results have shown that our classification model is capable of detecting some fresh malware. Finally, we also conduct some experiments to compare our approach with Yeganeh Safaei et al.'s approach [14], Allix et al.'s approach [12], Drebin [16] and most of anti-virus tools gathered in VirusTotal [17]. The results have confirmed that our classification model has a better performance in terms of accuracy than the others.

In summary, our contributions are as follows:

– We have proposed an approach to detecting Android malware based on deep learning, using two classic semantic representations in program analysis techniques, namely, control flow graph and data flow graph.
– We have conducted several experiments, which demonstrate that our approach is viable and effective to detect Android malware, and has a better performance than a number of existing anti-virus tools in terms of accuracy.

The remainder of this paper is organised as follows. Section 2 describes our approach, followed by the experimental results in Section 3. Section 4 presents the related work, followed by some concluding remarks in Section 5.

2 Approach

In this section, we present our approach CDGDroid, an effective approach to detecting Android malware based on deep learning, using control flow graph (CFG for short) and data flow graph (DFG for short). Figure 1 shows the framework of our approach, which consists of two phases: the *training* phase (marked by arrows with solid line) and the *testing* phase (marked by arrows with broken line). The training phase aims to train a classification model from an existing dataset containing normal applications and malware samples, and the testing phase uses the trained model to detect malware from new, unseen Android applications. In detail, we first use Apktool [18] to disassemble the applications in the given dataset and collect the *smali* files from each application. We then perform static analysis on these *smali* files to extract CFGs and DFGs, which are further encoded into matrices with known categories, yielding a training data set. Based on this training set, we train a classification model via CNN. Next, we perform the similar analysis on unseen Android applications to extract their feature matrices and then use the trained model to learn their categories. To conclude, our approach involves three tasks: (*i*) graph extracting; (*ii*) graph encoding; (*iii*) model training. In what follows, we depict each task of our approach in detail.

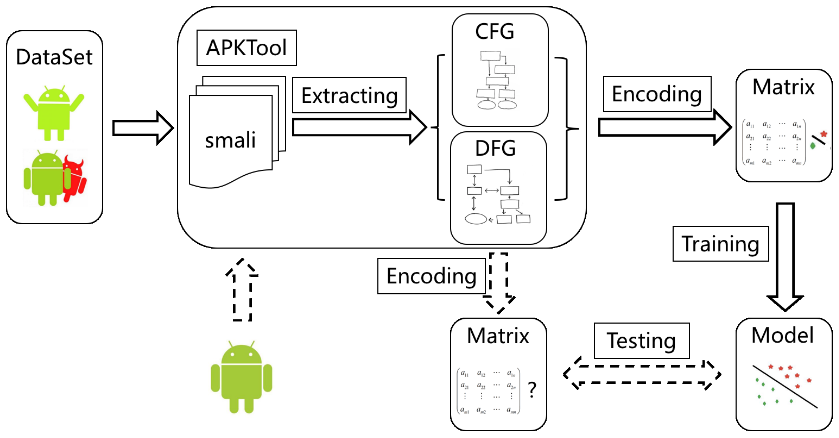

Fig. 1. Framework of Our Approach

2.1 Graph Extracting

This section is devoted to CFG and DFG extraction from an application in the instruction level, which consists of three steps: pre-processing, CFG extraction, and DFG extraction.

Pre-Processing. Android applications are distributed in markets in the form of APK. An APK is a compressed archive of Dalvik bytecode for execution, resources, assets, certificates, and an XML manifest file. Among them, the Dalvik bytecode for execution, namely, the file named *classes.dex*[5], will be extracted for further analysis.

For ease of extracting CFGs and DFGs, we leverage the disassembler Apktool [18] to disassemble the *dex* files. After disassembling, the *dex* files are converted to *smali* files, which give us the readable code in the *smali* language. We use *smali* code, instead of *Java* code, is because the disassembling is lossless in that the *smali* files support the full functionality of the original *dex* files.

CFG Extracting. There are several tools for generating CFGs for *smali* files, such as androguard and Smali-CFGs. Unfortunately, the CFGs generated by these existing tools are either lack of inter-procedural control flow, or not suitable for further analysis for us (*e.g.*, it is not easy to analyse CFG in the xgmml or PNG format). Therefore we implement CDGDroid with the CFG extracting based on the *smali* files.

To begin with, we give a definition of *graph*, which is used to describe both CFG and DFG.

Definition 1. *A graph G is a quadruple (N, E, S, F), where N is a finite set of nodes, $E \subseteq N \times N$ is a finite set of edges, $S \subseteq N$ is the set of starting nodes, and $F \subseteq N$ is the set of exiting nodes.*

[5] There may be several additional *dex* files with the name “classes*i*.dex” in large APKs.

Generally, a *smali* file contains the definition of a separate class, either a general class or an inner class, in the *Java* source code. So we construct the CFGs method by method for each *smali* file.

To do this, we first identify all the instructions in a method, yielding a graph *cfg* with the instructions as nodes and the first instruction as the starting node. This is different from existing tools, which take *block*s (*i.e.*, a straight-line piece of code without any jump instructions or jump targets) as basic nodes. Next, we complete this graph *cfg* by connecting the control flows (*e.g.*, a jump instruction and its targets) and identifying all the exiting nodes (*i.e.*, the reachable nodes without any out edges).

For inter-procedural analysis, we first construct a function call graph fcg by identifying the instructions starting with "invoke" or "execute"[6]. Based on this graph, we then connect the calling node with the start node of the callee method's CFG as well as the exiting nodes of the callee method's CFG with the successors of the calling node.

DFG Extracting. The DFG extracting is based on the CFG we extracted above. It is known that Dalvik is a register-based virtual machine, where most of the values are moved from memory into registers for access. So we will consider for DFG the data dependence relations between instructions via registers, including parameter registers.

Our construction of DFG is based on a variant of classic reaching definition analysis on *smali*, which is shown in Algorithm 1. This algorithm takes the CFG of a function f as input and then returns the reaching definition mapping D, which records the entry definitions (*i.e.*, *in*) and the exit definitions (*i.e.*, *out*) for each instruction. Note that, due to the inter-procedural analysis, we also take the initial definitions of parameters (*i.e.*, instructions starting with ".parameter") into account (Line 5).

Next, we extract the def-use relations as the edges of DFG, that is, if an instruction i uses a register r, whose value may come from the definition in the instruction j, then there is an edge from i to j. Algorithm 2 gives the detail of extracting. This algorithm takes the CFG of a function f as input and then returns the DFG for f, where Algorithm 1 is invoked to gather the def-use relations.

Similar to the CFG extracting, we also take the inter-procedural analysis into account via the function call graph *fcg*. In more detail, we connect the instructions that involved the definitions of arguments in the calling node with the start nodes of the callee method's DFG as well as the exiting nodes of the callee method's DFG with the special successor of the calling node (*i.e.*, an instruction starting with "move-result") if it exists.

2.2 Graph Encoding

It is straightforward to represent a graph as a matrix, such as the adjacency matrix. For a (simple) method, the adjacency matrix of the CFG is fine, but it could

[6] For simplicity, Java reflection, callbacks and multi-threading are not considered at present.

Algorithm 1 Reaching Definition Algorithm $RD(cfg)$

```
Input: CFG $cfg$ of a target function $f$
Output: the reaching definition mapping $D$
 1: for each node $n \in cfg.N$ do
 2:    $D(n).in = \emptyset$ and $D(n).out = \emptyset$
 3: end for
 4: for each starting node $e \in cfg.S$ do
 5:    enqueue $e$ in $q$ and add the initial definition of each parameter $p_i$ into $D(e).in$
 6: end for
 7: while $q \neq \emptyset$ do
 8:    $n =$ dequeue $q$
 9:    if $n$ is a definition with $r$ then
10:       $D(n).out = \{(r : n)\} \cup (D(n).in - \{(r : \_)\})$
11:    else
12:       $D(n).out = D(n).in$
13:    end if
14:    for all nodes $s$ in successors(n) do
15:       $(oi, ou) = D(s)$ and $D(s).in = D(s).in \cup D(n).out$
16:       if $D(s).in \neq oi$ and $s$ not in $q$ then
17:          enqueue $s$ into $q$
18:       end if
19:    end for
20: end while
21: return $D$
```

Algorithm 2 DFG Extracting Algorithm $DFG(cfg)$

```
Input: CFG $cfg$ of a target function
Output: DFG $dfg$ of the function
 1: $D = RD(cfg)$
 2: $dfg.N = cfg.N$, $dfg.E = \emptyset$, $dfg.S = \emptyset$ and $dfg.F = \emptyset$
 3: for each node $n \in cfg.N$ do
 4:    for each register $r$ used by $n$ do
 5:       for each definition $d$ of $r$ in $D(n).in$ do
 6:          $dfg.E = dfg.E \cup \{(d, n)\}$
 7:          if $d$ is the initial definition of a parameter then
 8:             $dfg.S = dfg.S \cup \{n\}$
 9:          end if
10:          if $n$ is the exiting node then
11:             $dfg.F = dfg.F \cup \{d\}$
12:          end if
13:       end for
14:    end for
15: end for
16: return $dfg$
```

be very large for an application, even a small one. This is mainly due to the large number of nodes. Similar to existing work [14], we abstract each instruction as its opcode, for example, the instruction "invoke-virtual v0, Ljava/lang/Object;->toString();" is abstracted as "invoke-virtual". In particular, we consider all the 222 opcodes in total listed in Android Dalvik-bytecode list [19].

In more detail, given a CFG (or DFG) g, we encode it into a matrix A with size 222×222 as follows: for each edge $(n_1, n_2) \in g.E$, we add the element $A[op(n_1)][op(n_2)]$ by 1, where A is initialised as a zero matrix with size 222×222 and $op(n)$ returns the opcode of the node n. Similarly, we accumulatively add all the encoded matrices of CFGs (resp. DFG) extracted from an application as its CFG matrix (resp. DFG matrix), denoted as A_{cfg} (resp. A_{dfg}). Moreover, due to the sparseness, we also add the matrix encoded from the control-flow (resp. data-flow) edges connected by the inter-procedural analysis (*i.e.*, the function call graph *fcg*) into A_{cfg} (resp. A_{dfg}). The resulting matrix is denoted as A_{scfg} (resp. A_{sdfg}), so as to differentiate from the matrix A_{cfg} (resp. A_{dfg}) above.

We also consider the combination of CFG and DFG. Firstly, as a program dependence graph, we combines these two graphs together into a graph. So the first mode, called the *vertical* one, is to combine these two graphs together via the matrix addition (denoted as $A_{cfg} + A_{dfg}$), that is, the vertical combination of A_{cfg} and A_{dfg} is a matrix A such that for each $i \in [1, 222]$ and $j \in [1, 222]$

$$A[i][j] = A_{cfg}[i][j] + A_{dfg}[i][j]$$

Secondly, we also would like to use them as different features, just like multi-views [20]. So the second mode, called the *horizontal* one, combines them via the matrix extension (denoted as $A_{cfg} \oplus A_{dfg}$). The resulting matrix A is of size 444×222 [7] instead, and satisfies that for each $i \in [1, 444]$ and $j \in [1, 222]$

$$A[i][j] = \begin{cases} A_{cfg}[i][j] & \text{if } 1 \leq i \leq 222 \\ A_{dfg}[i-222][j] & \text{otherwise} \end{cases}$$

2.3 Model Training

CNN is a new frontier in machine learning that has successfully been applied to analyse visual imagery (*i.e.*, matrix data). So we use CNN to train our model. In detail, we use the *Sequential* container to build our network model, which consists of 4 main layers, namely, a convolution layer with a reshape, a pooling layer and two fully connected layers, and uses the negative log likelihood criterion to compute a gradient. Note that, the reshape prior to convolution is used to reduce the number of parameters and thus save the training time. Figure 2 shows the structure of our network model.

3 Experiments

In this section, we conduct a series of experiments to evaluate our approach. Firstly, we conduct a set of cross-validation experiments to see the effectiveness

[7] The alternative extension with size 222×444 is fine as well.

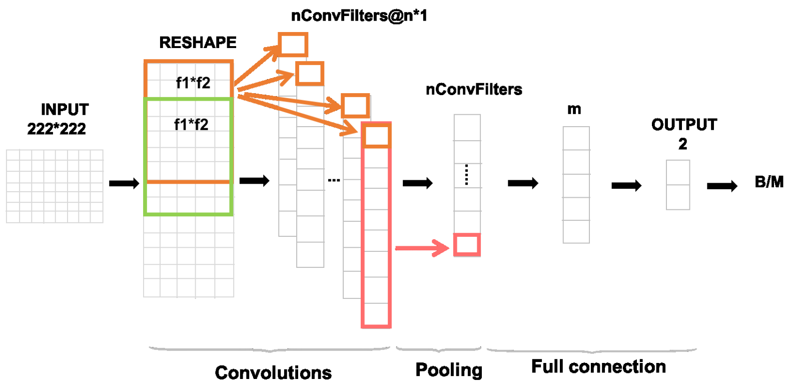

Fig. 2. Framework of Our Approach

of CFG and DFG in malware detection. Secondly, to test our approach's ability to detect unknown samples, we run our model on a dataset consisting of fresh samples. Finally, we also conduct some experiments to compare our approach with some existing Android malware detecting tools.

3.1 Dataset and Evaluative Criteria

We collect the samples mainly from four datasets, namely, Marvin [21], Drebin [16], VirusShare [22], and ContagioDump [23]. The Marvin dataset contains a training set (with 50501 benign samples and 7406 malware samples) and a testing set (with 25495 benign samples and 3166 malware samples). The other three datasets, Drebin, VirusShare and ContagioDump[8], contain only malware samples, with 5560, 11080 and 1150 samples, respectively. We also collect 1771 applications from Mi App Store [24], which pass the detecting of most anti-virus tools gathered in VirusTotal [17] and thus are considered as benign samples.

To quantitatively validate the experimental results, we use the following performance measures. *Accuracy* is the most intuitive performance measure and it is simply a ratio of correctly predicted observation to the total observations. *Precision* is the ratio of correctly predicted positive observations to the total predicted positive observations, and *Recall* is the ratio of correctly predicted positive observations to all observations in actual class. *F1 score* is the weighted average of *Precision* and *Recall*, that is, $(2 \cdot Precision \cdot Recall)/(Precision + Recall)$. *AUC* is the area under ROC curve, which is (arguably) the best way to summarize its performance in a single number. Intuitively, the higher the measures above, the better the classifier.

[8] Only the malware samples whose creation dates are in 2018 are collected.

3.2 Experiments on Different Features

In this section, we first conduct experiments to evaluate how CFG and DFG contribute to the effectiveness of malware detection. We then run experiments to see how the combination modes affect the malware detection.

CFG and DFG. We separately use the feature matrices A_{cfg}, A_{scfg}, A_{dfg} and A_{sdfg} to train the classification model on the dataset consisting of the training set of Marvin, Drebin and VirusShare, where *10-fold cross validation* is employed. In addition, we also consider traditional CFGs [9] based on *blocks* and encode them into matrices (called as A_{tcfg}) in a similar way, where all the blocks without jumps are abstracted as a special node. More specifically, only nodes involving "control" are preserved in A_{tcfg}, yielding a matrix with size 32×32 (*i.e.*, 31 "control" opcodes and 1 special node for the other nodes). The experimental results are given in Table 1.

Table 1. Results on Different Feature Matrices

Feature	Precision	Accuracy	Recall	F1 Score	AUC
A_{cfg}	99.833%	99.400%	94.691%	97.195%	0.999
A_{scfg}	100.000%	99.194%	92.704%	96.214%	0.999
A_{dfg}	99.869%	99.613%	96.620%	98.218%	0.999
A_{sdfg}	99.835%	99.470%	95.370%	97.552%	0.999
A_{tcfg}	99.842%	95.568%	59.981%	74.941%	0.993

The results show that all the features are effective in detecting malware, with *accuracy* larger than 95.5%, *precision* larger than 99.8%, and *AUC* larger than 0.99. Compared with CFG, the feature DFG performs better, both for intra-procedural analysis (1.053% higher in *F1 score*) and inter-procedural analysis (1.391% higher in *F1 score*). A possible reason is that DFGs are built on CFGs such that DFGs would, in some sense, contain some "control flow" information. Rather surprisingly, the graphs with inter-procedural analysis perform worse than the ones without. More specifically, the *F1 score* of A_{cfg} is 1.020% higher than the one of A_{scfg}, although the *precision* of A_{cfg} is better than the one of A_{scfg}; and A_{dfg} is 0.683% better than A_{sdfg} in term of *F1 score.* There are two possible reasons behind this: (1) the ignoring of callbacks and multi-threading makes the function call graphs incomplete; (2) accumulating the matrices extracted from inter-procedural analysis and the ones from intra-procedural analysis together might have lost the differences between them, thus make against the model.

In addition, we have found that the *AUC* of A_{tcfg} is 0.993, which is pretty high, and thus also demonstrates that "control flow" information is capable of facilitating detect malware. Moreover, the *precision* and the *accuracy* of A_{tcfg}

[9] For convenience, we do not consider the inter-procedural analysis for traditional CFGs, since the instructions for method calling are abstracted as the special node.

are quite close to the ones of A_{cfg} and A_{dfg}, although the *recall* and *F1 score* are not so high. We may take *block*s into account to improve A_{tcfg} as shown in [12], which is left as future work. As A_{tcfg} is much simpler than A_{cfg} and A_{dfg}, we believe that A_{tcfg} can be used as a feature of models on-device.

Combination Modes. In these experiments, we use the feature matrix $A_{cfg} + A_{dfg}$ and $A_{cfg} \oplus A_{dfg}$ to train the model on the same dataset as above, respectively. Note that, as shown in the experiments above, the graphs without inter-procedural analysis perform better, so we do not consider A_{scfg} and A_{sdfg} here. The experimental results are shown in Table 2.

Table 2. Results on Different Modes

Feature	Precision	Accuracy	Recall	F1 Score	AUC
$A_{cfg} + A_{dfg}$	99.770%	99.536%	96.020%	97.859%	0.999
$A_{cfg} \oplus A_{dfg}$	99.903%	99.721%	97.568%	98.722%	0.999

From the results, we can see that the horizontal combination $A_{cfg} \oplus A_{dfg}$ performs better than both A_{cfg} and A_{dfg}, and thus the horizontal combination can improve the detection. While the vertical combination $A_{cfg} + A_{dfg}$ performs better than A_{cfg} but worse than A_{dfg}. That is to say, the vertical combination may make against the detecting model. Similar to inter-procedural analysis, a possible reason is that the vertical combination, *i.e.*, adding CFGs and DFGs together, could lose their differences.

3.3 Experiments on Unknown Samples

To test the viability of the proposed approach to detect unknown samples, we run our model trained with the feature matrix $A_{cfg} \oplus A_{dfg}$ respectively on two datasets: the first one comes from the testing set of Marvin, and the second one consists of the new malware samples from ContagioDump, whose creation date is in 2018. The experimental results are shown in Table 3.

Table 3. Results on Other Dataset

Dataset	Precision	Accuracy	Recall	F1 Score
Marvin	99.649%	99.822%	98.737%	99.191%
ContagioDump	100.000%	72.870%	72.870%	84.301%

It can be seen from the results that the proposed CDGDroid is capable of detecting some fresh malware. In detail, CDGDroid performs on the testing set of Marvin quite well, with the *precision* 99.649%, the *accuracy* 99.822%, the *recall* 98.737% and the *F1 score* 99.191%. And for ContagioDump dataset, there are 72.870% malware samples that can be detected by CDGDroid. ContagioDump comprises only malware samples, so the *precision* of detection is 100%. Compared with the testing set of Marvin, the performance of CDGDroid is a little worse. One main reason is that the samples in ContagioDump are collected *later* than the ones in the training set (*i.e.*, Marvin, Drebin and VirusShare), that is, the samples in ContagioDump are *genuinely new*.

3.4 Comparison against Malware Detecting Tools

In this section, we present experiments to compare our approach with some recent tools, namely, Yeganeh Safaei et al.'s approach [14] (based on CNN), Allix et al.'s approach [12] (using CFG), Drebin [16] (using 8 other features), and VirusTotal [17] (gathering a variety of anti-virus tools).

DODroid. Yeganeh Safaei et al. [14] recently proposed an Android malware detection system based on a deep convolutional neural network, using the raw opcode sequence as features. We dub this system "DODroid" (Deep Opcode). As both CDGDroid and DODroid use CNN to build the classification model, we conduct experiments to compare CDGDroid against DODroid. In detail, we use the same training dataset (*i.e.*, the training set of Marvin) to train both CDGDroid and DODroid, and then use the same testing dataset (*i.e.*, the testing set of Marvin) to test these two models. The experimental results are shown in Table 4.

Table 4. Comparison against DODroid

Tool	**Precision**	**Accuracy**	**Recall**	**F1 Score**
CDGDroid	99.903%	99.721%	97.568%	98.722%
DODroid	98.396%	99.067%	93.137%	95.695%

It can be seen from the results that CDGDroid outperforms DODroid. Regardless of the slight differences of two CNN models, the results also show that CFG and DFG are more effective than opcodes in malware detection, that is, *control flows* and *data flows* can help in detecting malware.

CSBD. Allix et al. [12] proposed another scalable approach using structural features, namely textual representations of the CFGs. Here we compare our approach against the re-implementation of Allix et al.'s approach from [25], where Random Forest is used to train the classifier and this approach is referred as CFG-Signature Based Detection (CSBD). So we also refer this approach as CS-BD here. The experiments are similar to the ones of DODroid. The experimental results are shown in Table 5.

We can see that CSBD has a better *recall*, while CDGDroid gets a better *precision*. A main reason is that CSBD takes *block*s of CFGs as features, while CDGDroid focuses on control flow and data flow information, plus a simple *block* information (*i.e*, the adjacency information of nodes). In short, CDGDroid gets a better *F1 score* than CSBD, so we conclude that CDGDroid performs better than CSBD.

Table 5. Comparison against CSBD

Tool	Precision	Accuracy	Recall	F1 Score
CDGDroid	99.903%	99.721%	97.568%	98.722%
CSBD	92.151%	99.033%	99.747%	95.799%

Drebin. Drebin [16] is a lightweight Android malware detecting tool based on SVM, which uses 8 different types of features, namely, hardware components, requested permissions, app components, filtered intents, restricted API calls, used permissions, suspicious API calls, and network addresses. We also conduct experiments to compare against Drebin, where we use the re-implementation of Drebin from [25] as well. The experiments are performed on the malware samples from Drebin and the benign samples from Marvin and from Mi App Store. Table 6 gives the experimental results.

Table 6. Comparison against Drebin

Tool	Precision	Accuracy	Recall	F1 Score
CDGDroid	99.781%	99.870%	98.273%	99.021%
Drebin	91.000%	99.123%	96.000%	94.000%

The results show that CDGDroid performs better than Drebin, which also indicates that the features we consider (*i.e.* CFG and DFG) are quite effective in malware detection, with respective to the 8 features used in Drebin.

VirusTotal. VirusTotal [17] is a free online malware detecting website, which gathers a variety of anti-virus tools. For comparison, we design a crawler to automatically upload the samples in the testing set of Marvin into VirusTotal for further detecting by those anti-virus tools, which lasts almost one week. The results are shown in Table 7, where those tools with too few responds from VirusTotal are filter out.

From the results, we can see that our tool CDGDroid outperforms most of anti-virus tools. In particular, our tool CDGDroid gets the best *accuracy* (99.822%) and the best *F1 score* (99.191%). Although there are 3 (resp. 12) tools having a better *precision* (resp. *recall*) than CDGDroid, the gaps of *precision* (resp. *recall*) between CDGDroid and these tools are quite small.

4 Related Work

Over the past decade, there are a lot of research work for Android malware detection. Here we only review some related and recent ones, namely, graph based detection and deep learning based detection.

Graph based Detection. Sahs and Khan [9] proposed a machine learning-based system which extracts features from control flow graphs of applications. Al-

Table 7. Comparison against Anti-Virus Tools in VirusTotal

Tool	Precision	Accuracy	Recall	F1 Score	Tool	Precision	Accuracy	Recall	F1 Score
CDGDroid	99.649%	99.822%	98.737%	99.191%	Kaspersky	98.434%	99.789%	99.683%	99.050%
Avast	98.509%	99.746%	99.201%	98.850%	DrWeb	96.427%	99.370%	98.025%	97.220%
Jiangmin	97.682%	99.318%	96.134%	96.900%	Qihoo-360	96.013%	99.490%	97.556%	96.780%
GData	94.057%	99.252%	99.553%	96.730%	Emsisoft	94.050%	99.250%	99.550%	96.720%
TrendMicro	97.900%	99.198%	94.694%	96.270%	Sophos	91.998%	98.956%	99.169%	95.450%
BitDefender	94.049%	98.831%	95.431%	94.740%	Alibaba	89.973%	99.121%	98.793%	94.180%
F-Secure	90.808%	98.480%	96.151%	93.400%	QuickHeal	88.296%	98.433%	99.055%	93.370%
NOD32	87.182%	98.277%	99.621%	92.990%	Ikarus	86.414%	98.250%	99.743%	92.600%
Arcabit	89.076%	98.607%	92.393%	90.700%	K7GW	85.283%	98.414%	96.400%	90.500%
Tencent	83.100%	98.447%	98.843%	90.290%	Comodo	92.557%	97.777%	86.808%	89.590%
Symantec	92.101%	97.776%	87.198%	89.580%	VBA32	99.960%	97.791%	80.152%	88.970%
Fortinet	85.522%	97.414%	92.254%	88.760%	AVware	77.629%	97.791%	99.627%	87.260%
Avira	77.929%	97.528%	96.038%	86.040%	Antiy-AVL	89.085%	96.807%	81.067%	84.890%
AegisLab	76.108%	97.182%	94.857%	84.450%	Microsoft	99.956%	96.990%	72.820%	84.260%
NANO	70.888%	96.648%	97.934%	82.240%	Cyren	65.584%	95.868%	99.737%	79.130%
VIPRE	93.002%	95.590%	65.046%	76.550%	F-Prot	78.086%	94.639%	71.920%	74.880%
McAfee	80.799%	93.392%	53.167%	64.130%	AVG	81.474%	93.337%	51.707%	63.260%
AhnLab-V3	85.445%	92.747%	49.817%	62.940%	McAfee-GW	92.244%	93.216%	42.424%	58.120%
TotalDefense	99.810%	92.399%	33.228%	49.860%					

lix et al. [12] devised several machine learning classifiers that rely on a set of features which are textual representations of the control flow graphs of applications. DroidMiner [10] digs malicious behavioral patterns from a two-level behavioural graph representation built on control-flow graphs and call graphs. AppContext [26] extracts the contextual information of security-sensitive activities along with structural information through reduced inter-procedure control-flow graphs, and CWLK [25] is a similar approach, which extracts the information through call graphs and inter-procedural control-flow graphs. DroidOL [13] is an online machine learning based framework, which extracts features from inter-procedural control-flow sub-graphs. MKLDroid [20] integrates context-aware multiple views to detect Android malware, where all views are built from inter-procedural control flow graphs. However, most of these approaches only consider control flow properties, leaving data flow properties out of consideration.

Data flow analysis is also adopted in malware detection. Flowdroid [3] and Amandroid [4] are two state-of-the-art data flow analysis tools for Android. Andriatsimandefitra and Tong [27] proposed to use system flow graphs, constructed from the log of an information flow monitor, to characterise malware samples. DroidSIFT [28] takes a weighted contextual (security-related) API dependency graph as semantics feature sets and use graph similarity metrics to detect malware. DroidADDMiner [11] is a machine learning based system that extracts features based on data dependency between sensitive APIs. However, all these tools rely on heavyweight data flow analyses.

There are some approaches that take both control flow and data flow properties into account. Apposcopy [29] and ASTROID [30] detect Android malware via signature matching on program graphs, including certain control- and data-flow properties. CASANDRA [31] extracts features from contextual API dependency graphs, containing structural information and contextual information. Different from these approaches, we use deep learning to build our classification model.

Some other graphs are used to detect Android malware as well, such as function call graphs [32–34], permission event graphs [35], component topology graph [36].

Deep Learning Based Detection. Droid-Sec [37] and DroidDetector [38] used the deep belief network (DBN) to build the classification model, taking required permission, sensitive API and dynamic behaviour as features. Droid-deep [39] built the model by DBN as well, but used some more features (e.g., actions and components). DroidDelver [40] and DroidDeepLearner [41] are another two models built on DBN, where permissions and API calls were taken as features. Mclaughlin [14] designed the detection systems by Convolutional Neural Network (CNN), using opcode sequences as features. Nix and Zhang [42] and MalDozer [43] also built the system by CNN, but used system API call sequences as features. Deep4MalDroid [44] is a deep learning framework (*i.e.*, Stacked AutoEncoders) resting on the system call graphs extracted by dynamic analysis from Android applications. Nauman et al. [45] applied several deep learning models including fully connected, convolutional and recurrent neural networks as well as autoencoders and deep belief networks to detect Android malware, using the eight features proposed in [16]. DeepFlow [46] identified malware directly from the data flows in the Android application based deep learning.

Most of these approach consider neither control flow nor data flow information (except DeepFlow), while our approach takes both control and data flow graphs into account.

5 Conclusion

In this work, we have proposed an Android malware detection approach based on CNN, using control flow graph (CFG) and data flow graph (DFG). To evaluate the proposed approach, we have carried out some interesting experiments. Through experiments, we have found that the classification model with the horizontal combination of CFG and DFG as features performs the best. The experimental results have also demonstrated that our classification model is capable of detecting some fresh malware, and has a better performance than Yeganeh Safaei et al.'s work, Drebin and most of anti-virus tools gathered in VirusTotal.

As for future work, we may consider a better function call graph to improve the approach. We can use other program graphs, such as program dependence graphs, to train the model. We can also leverage N-Gram to extract program traces with length N as features. More experiments on malware anti-detecting techniques (*i.e.*, obfuscation techniques) are under consideration.

Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful comments. This work was partially supported by the National Natural Science Foundation of China under Grants No. 61502308 and 61772347, Science and Technology Foundation of Shenzhen City under Grant No. JCYJ20170302153712968, Project 2016050 supported by SZU R/D Fund and Natural Science Foundation of SZU (Grant No. 827-000200).

References

1. *Report from IDC.* http://www.idc.com/promo/smartphone-market-share/os.
2. *Report from G DATA*, 2017. https://www.gdatasoftware.com/blog/2017/04/29712-8-400-new-android-malware-samples-every-day.
3. S. Arzt and et al. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI '14*, pages 259–269, 2014.
4. F. Wei, S. Roy, and X. Ou. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *CCS '14*, pages 1329–1341, 2014.
5. W. Enck and et al. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI '14*, pages 393–407, 2014.
6. W. Enck, M. Ongtang, and P. Mcdaniel. On lightweight mobile phone application certification. In *CCS '09*, pages 235–245, 2009.
7. A. Felt and et al. Android permissions demystified. In *CCS '11*, pages 627–638, 2011.
8. M. Grace and et al. Riskranker: scalable and accurate zero-day android malware detection. In *MobiSys '12*, pages 281–294, 2012.
9. J. Sahs and L. Khan. A machine learning approach to android malware detection. In *EISIC '12*, pages 141–147, 2012.
10. C. Yang and et al. DroidMiner: Automated Mining and Characterization of Fine-grained Malicious Behaviors in Android Applications. In *ESORICS '14*, pages 163–182, 2014.
11. Y. Li and et al. Detection, Classification and Characterization of Android Malware Using API Data Dependency. In *SecureComm '15*, pages 23–40, 2015.
12. K. Allix and et al. Empirical assessment of machine learning-based malware detectors for android. *Empirical Software Engineering*, 21(1):183–211, 2016.
13. Narayanan A, Y. Liu, L. Chen, and J. Liu. Adaptive and scalable android malware detection through online learning. In *IJCNN '16*, pages 157–175, 2016.
14. N. Mclaughlin and et al. Deep android malware detection. In *CODASPY '17*, pages 301–308, 2017.
15. Sen Chen and et al. Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach. *Computers and Security*, 73:326–344, 2017.
16. D. Arp and et al. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *NDSS '14*, 2014.
17. *VirusTotal.* https://www.virustotal.com.
18. R. Wiśniewski and C. Tumbleson. *Apktool: A tool for reverse engineering Android apk files.* https://ibotpeaches.github.io/Apktool/.
19. *Dalvik Bytecode.* https://source.android.com/devices/tech/dalvik/dalvik-bytecode.
20. A. Narayanan, M. Chandramohan, L. Chen, and Y. Liu. A multi-view context-aware approach to android malware detection and malicious code localization. *Empirical Software Engineering*, pages 1–53, 2017.
21. M. Lindorfer, M. Neugschwandtner, and C. Platzer. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In *ComSAC '15*, pages 422–433, 2015.
22. *VirusShare.* https://virusshare.com/.
23. *Contagiodump.* http://contagiodump.blogspot.com/.
24. *Mi App Store.* https://dev.mi.com/en.

25. A. Narayanan and et al. Contextual weisfeiler-lehman graph kernel for malware detection. In *IJCNN '16*, pages 4701–4708, 2016.
26. W. Yang and et al. Appcontext: differentiating malicious and benign mobile app behaviors using context. In *ICSE '15*, pages 303–313, 2015.
27. R. Andriatsimandefitra and V. Tong. Capturing android malware behaviour using system flow graph. In *NSS '14*, pages 534–541, 2014.
28. M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *CCS '14*, pages 1105–1116, 2014.
29. Y. Feng, S. Anand, L. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *FSE '14*, pages 576–587, 2014.
30. Y. Feng and et al. Automated synthesis of semantic malware signatures using maximum satisfiability. *CoRR*, abs/1608.06254, 2016.
31. A. Narayanan, M. Chandramohan, L. Chen, and Y. Liu. Context-aware, adaptive and scalable android malware detection through online learning (extended version). *CoRR*, abs/1706.00947, 2017.
32. H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural Detection of Android Malware Using Embedded Call Graphs. In *AISec '13*, pages 45–54, 2013.
33. Y. Du, J. Wang, and Q. Li. An android malware detection approach using community structures of weighted function call graphs. *IEEE Access*, PP(99):1–1, 2017.
34. M. Fan and et al. Frequent subgraph based familial classification of android malware. In *ISSRE '16*, pages 24–35, 2016.
35. K. Chen and et al. Contextual policy enforcement in android applications with permission event graphs. *Heredity*, 110(6):586, 2013.
36. T. Shen and et al. Detect android malware variants using component based topology graph. In *TrustCom '14*, pages 406–413, 2014.
37. Z. Yuan, Y. Lu, Z. Wang, and Y. Xue. Droid-Sec: Deep Learning in Android Malware Detection. In *SIGCOMM '14*, pages 371–372, 2014.
38. Z. Yuan, Y. Lu, and Y. Xue. Droiddetector: Android malware characterization and detection using deep learning. *Tsinghua Science and Technology*, 21(1):114–123, 2016.
39. X. Su, D. Zhang, W. Li, and K. Zhao. A deep learning approach to android malware feature learning and detection. In *TrustCom '16*, pages 244–251, 2016.
40. S. Hou, A. Saas, Y. Ye, and L. Chen. DroidDelver: An Android Malware Detection System Using Deep Belief Network Based on API Call Blocks. In *WAIM '16*, pages 54–66, 2016.
41. Z. Wang, J. Cai, S. Cheng, and W. Li. Droiddeeplearner: Identifying android malware using deep learning. In *Sarnoff '16*, pages 160–165, 2016.
42. R. Nix and J. Zhang. Classification of android apps and malware using deep neural networks. In *IJCNN '17*, pages 1871–1878, 2017.
43. E. Karbab, M. Debbabi, A. Derhab, and D. Mouheb. Maldozer: Automatic framework for android malware detection using deep learning. *Digital Investigation*, 24:S48–S59, 2018.
44. S. Hou, A. Saas, L. Chen, and Y. Ye. Deep4maldroid: A deep learning framework for android malware detection based on linux kernel system call graphs. In *WIW '17*, pages 104–111, 2017.
45. M. Nauman, T. Tanveer, S. Khan, and T. Syed. Deep neural architectures for large scale android malware analysis. *Cluster Computing*, pages 1–20, 2017.
46. D. Zhu and et al. Deepflow: Deep learning-based malware detection by mining android application for abnormal usage of sensitive data. In *ISCC '17*, pages 438–443, July 2017.