Android Malware Family Classification and Characterization Using CFG and DFG

Zhiwu Xu*†, Kerong Ren* and Fu Song‡
*College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China
†National Engineering Laboratory for Big Data System Computing Technology, Shenzhen University, Shenzhen, China
‡School of Information Science and Technology, ShanghaiTech University, Shanghai, China

***Abstract*—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. Recently, many approaches and tools to analyze Android malware have been proposed to protect legitimate users from the threat. However, most approaches focus on malware detection, while only a few of them consider malware classification or malware characterization. In this paper, we propose an extension of CDGDroid to classifying and characterizing Android malware families automatically. We first perform static analysis used in CDGDroid to extract control-flow graphs and data-flow graphs on the instruction level. Then we encode the graphs into matrices, and use them to build the family classification models via deep learning. For family characterization, we extract the n-gram sequences from the graphs, which are filtered according to the weights of the classification model built for the target family. And then we construct a vector space model and select the top-k sequences as a characterization of the target family. We have conducted some experiments to evaluate our approach and have identified that the family classification model taking the horizontal combination of CFG and DFG as features offers the best performance in terms of accuracy among all the models. Compared with CDGDroid, Drebin and many anti-virus tools gathered in VirusTotal, our family classification model gives a better performance. Finally, We have also conducted experiments on family characterization, and the experimental results have shown that our characterization can capture the malicious behaviors of the testing families.**

***Index Terms*—Android malware, malware family, malware characterization, static analysis, deep learning**

I. Introduction

As reported by IDC [1], Android is the most popular platform for mobile devices, with almost 86.8% of the market share in the third quarter of 2018. Unfortunately, the increasing adoption of Android comes with the growing prevalence of Android malware. A report from security firm G DATA [2] shows that around 3.2 million new malware apps up to the end of the third quarter of 2018. 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.

Over the past decade, many approaches and tools to analyze Android malware have been proposed to protect legitimate users from the threat, which can be summarized into three categories, namely, malware detection, malware classification and malware characterization. Malware detection aims to identity whether a given application is malware or not, which can protect the users directly; malware classification tries to group malware samples into families or identity the family for a malware sample, which helps to organize and analyze malware samples; and malware characterization is to characterize the underlying malicious behaviors or patterns for a malware family (or a malware sample), which will benefit the malware detection and classification. However, it is pity that most approaches focus on malware detection [3]–[6], while only a few of them consider malware classification or malware characterization. Moreover, these existing classification or characterization approaches either use program analysis [7]–[10], which may be too heavyweight to use in practice, or use machine learning with syntax features [11]–[14], which could be too simple to characterize malware.

In this paper, we propose an approach to classifying and characterizing Android malware families automatically, wherein two classic semantic representations of programs, namely, control-flow graph (CFG for short) and data-flow graph (DFG for short), are used as features, under the assumption that the given malware samples have been identified correctly by any malware detection toolkit such as CDGDroid [15]. Generally, a CFG reflects what a program intends to behave (e.g., opcodes) as well as how it behaves (e.g., possible execution paths), such that malware behavior patterns can be captured easily by this feature. For example, Geinimi samples share the similar control flow graphs. On the other hand, a DFG 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 classification approach is an extension of CDGDroid, which is an effective approach to detecting Android malware. Specifically, we perform static analysis used in CDGDroid to extract CFGs and DFGs on the instruction level for Android applications, and then abstract the graphs into weight graphs, where the nodes are opcodes or the invoke opcode with a permission and the weight of an abstracted edge denotes the number of the original edges that are abstracted into it. The main extension lies in that the permissions that an invoke instruction requests are considered during the graph abstraction. Next, we encode the weight graphs into matrices and build a family classification model via deep learning, which can be used to analyze new, unseen malware samples. Similar to CDGDroid, two combination modes of control flow graph and data flow graph are considered: two graphs are combined either via the matrix addition (called the *vertical* mode) or via the matrix extension (called the *horizontal* mode).

To characterize malware samples of a target family, we extract the n-gram sequences from the paths of the graphs, which are then filtered according to the weights of the classification model that are built for the target family. Based on the sequences, we construct a vector space model and select the top-k sequences as a characterization of the target family.

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 family classification. We have found that the classification model taking the horizontal combination of CFG and DFG as features performs the best, with the average accuracy 94.71% for all families. Next, we conduct some experiments to compare our approach with CDGDroid [15], Drebin [11] and most of anti-virus tools gathered in VirusTotal [16]. The results have confirmed that our family classification model has a better performance than the others in terms of accuracy. Finally, we also conduct some experiments on family characterization, and the experimental results have shown that our characterization can capture the malicious behaviors of the testing families.

In summary, our contributions are as follows:

- We have proposed an approach to classifying and characterizing Android malware families using CFGs and DFGs as features.
- We have conducted several experiments, which demonstrate that our approach is viable and effective to Android malware family classification, and has a better performance than a number of existing anti-virus tools in terms of accuracy.
- We have also conducted some experiments on family characterization, which shows that our characterization can capture the malicious behaviors of the testing families.

The remainder of this paper is organized as follows. Section II describes our approach, followed by the experimental results in Section III. Section IV presents the related work, followed by some concluding remarks in Section V.

II. Approach

In this section, we present our approach to classifying and charactering Android malware families using CFGs and DFGs as features. Figure 1 shows the framework of our approach, which consists of three tasks: *static analysis*, *family classification* and *family characterization*. Firstly, the static analysis task is meant to extract CFGs and DFGs from Android applications (APK for short) and then to abstract them into weight graphs. Secondly, the classification task aims to build a family classification model (or a binary classification for a target family) from an existing Android malware dataset based on deep learning, which can then be used to classify Android malware samples detected by any malware detection toolkit, such as CDGDroid [15]. Finally, the characterization task tries to mine the possible malicious behaviors on the opcode level for a target family, guiding by the corresponding classification

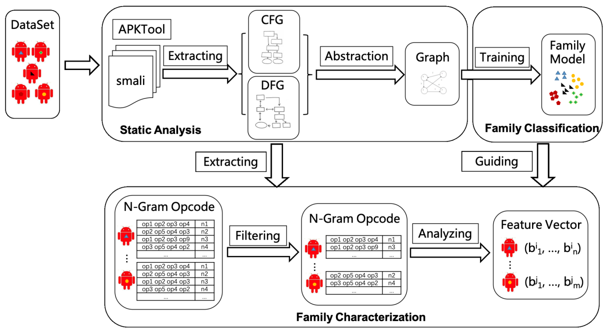


Fig. 1. Framework of Our Approach

model built for the target family. In what follows, we depict each task of our approach in detail.

A. Static Analysis

To begin with, we give the definitions of *graph*, which is used to describe both CFGs and DFGs, and *weight graph*, which is used to describe the abstractions of CFGs and DFGs.

Definition 1. *A graph G is a 4-tuple (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.*

Definition 2. *A weight graph G is a 5-tuple (N, E, S, F, W), where N, E, S, F are the same to the ones in Definition 1 and W is a weight mapping from edges E to natural numbers $\mathcal{N}$.*

Our static analysis is an extension of the analysis used in CDGDroid. We first perform the analysis in CDGDroid on the given APKs to extract CFGs and DFGs, where nodes of CFGs and DFGs are the instructions, rather than the basic blocks. Due to space limitation, the details are not presented here. Interested readers can refer to [15].

Generally, the purpose of a permission is to protect the privacy of an Android user. APKs must request permission to access sensitive user data (such as contacts and SMS), as well as certain system features (such as camera and internet). So the required permissions of an *invoke* instruction can reflect its intention and behavior, and thus can help in classifying android malware as well as characterizing the malicious behaviors in Section II-C. Hence, different from CDGDroid, we consider the permissions that an invoke instruction requests as well, when we perform the graph abstraction.

The detail of graph abstraction is shown in Algorithm 1, which takes a CFG or DFG as input and returns a weight graph, where the nodes are opcodes or the invoke opcode with a permission, and the weight of an abstracted edge denotes the number of the edges in the original graphs that are abstracted into it. The algorithm simply iterates the nodes and edges in the given graph by performing the instruction abstraction IA and accumulates the number of abstracted edges, which is almost the same as the one in [15], except the instruction abstraction IA.

The instruction abstraction IA is given in Algorithm 2, which takes any instruction as input and returns a singleton set or a set of the invoke opcode with different permissions. The algorithm first checks whether the instruction is an invoke one. If it is, then the algorithm analyzes the possible permission set that the invoke instruction requests, which can be done via Pscout [17]. Then the algorithm returns a singleton set of the invoke opcode if the permission set is empty, or a set of the invoke opcode associating with each permission in the set otherwise. If it is not an invoke instruction, the algorithm returns a singleton set of its opcode. While in [15], IA simply returns the opcode of the input instruction.

Algorithm 1 Graph Abstraction $GA(G)$

```
Input: a graph $G$
Output: a weight $G_a$
1: let $G_a = (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$
2: $G_a.N = \bigcup_{n \in G.N} IA(n)$
3: $G_a.S = \bigcup_{n \in G.S} IA(n)$
4: $G_a.F = \bigcup_{n \in G.F} IA(n)$
5: for $(n_1, n_2) \in G.E$ do
6:     for $n_1' \in IA(n_1)$ and $n_2' \in IA(n_2)$ do
7:         $G_a.E = G_a.E \cup \{(n_1', n_2')\}$
8:         inc $G_a.W[(n_1', n_2')]$
9:     end for
10: end for
11: return $G_a$
```

Algorithm 2 Instruction Abstraction $IA(i)$

```
Input: an instruction $i$
Output: a singleton set or a set of the invoke opcode with different permissions
1: if $i$ is an invoke instruction then
2:     let $c$ and $m$ be the invoked class and method of $i$
3:     let $pl$ be the required permission set of $c.m$
4:     if $pl$ is empty then
5:         return $\{invoke\}$
6:     else
7:         return $\{(invoke, p) \mid p \in pl\}$
8:     end if
9: else
10:     return {the opcode of $i$}
11: end if
```

B. Family Classification

The next step is to encode the weight graphs into adjacency matrices, which yields a training data set. In particular, we consider all the 222 opcodes in total listed in Android Dalvik-bytecode list [18], and the 91 permissions in total (including 29 protection normal permission, 36 protection signature permission and 26 dangerous permissions) listed in Android permission overview [19]. That is to say, the size of the encoded matrix is 313. Moreover, similar to CDGDroid, two combination modes of CFG and DFG are considered, namely, the vertical mode (denoted as $cfg + dfg$) and the horizontal model (denoted as $cfg \oplus dfg$). In the vertical mode, like constructing a program dependence graph, CFG and DFG are combined via the matrix addition; while in the horizontal model, they are combined via the matrix extension, just as they are treated as different views. Finally, we build the classification models based on neural network, whose network structures are similar to the one used in CDGDroid, except the number of labels, that is the size of the last layer.

C. Family Characterization

This section is devoted to mine the malicious behaviors, namely, the n-gram sequences of opcodes, for a target Android malware family, which consists of three steps: behavior extraction, behavior filtering, and behavior analysis.

1) Behavior Extraction: An instruction, in particular, the opcode, specifies the operation to be performed. We call a sequence of instructions or opcodes as a behavior. Instead of from the raw instruction text, here we extract the opcode sequences from the CFGs or DFGs of a malware sample, via n-gram. The extracting procedure is given in Algorithm 3. In detail, given a CFG or DFG, we first extract all the possible paths of the graph and then collect the n-gram sequences for all the paths, yielding the behavior set for the malware sample. Likewise, to reduce the size of the behavior set, we perform the instruction abstraction IA on the instruction sequences as well. In addition, during extracting we also count the frequencies of the opcode sequences.

Algorithm 3 Graph nGram $GnG(G, n)$

```
Input: a graph $G$
Output: an opcode sequence set
1: let $paths$ be all the paths of $G$ and $res = \emptyset$
2: for $p \in paths$ do
3:     let $seqs = nGram(p, n)$
4:     for $seq \in seqs$ do
5:         let $seqs\_ia = \{r_1 \ldots r_{|seq|} \mid r_i \in IA(seq[i])\}$
6:         for $seq\_ia \in seqs\_ia$ do
7:             $res \cup= \{seq\_ia\}$
8:             inc $nums[seq\_ia]$
9:         end for
10:     end for
11: end for
12: return $res$
```

2) Behavior Filtering: To reduce the size of the opcode sequence set further, we filter these sequences with respect to the binary classification built for the target family (in the family classification task). To do that, we first approximately calculate the weights of the target classification model, which reflect how each unit of the input matrix contributes to the classification. Then we take the units of top-k weights as a reference, and filter out the extracted opcode sequences that do not contain these selected units for the target family. Indeed, each unit of the input matrix is a 2-gram opcode sequence. And we believe that the selected 2-gram or n-gram sequences contribute to the interpretation of the family classification models.

Weight Calculation. We calculate the weights back-forward and layer by layer for the target classification model. For simplicity, we only consider the weight of the highest degree for each layer. The last layer of a binary model is always a vector with size 2, and the first value of the vector always denotes the possibility of the input belonging to the target class. So we set the weight for the last layer as $[1, 0]$. Without loss of generality, we assume that the last i-th layer is a matrix of size $m_i \times n_i$ and its weight matrix contributing to the classification is W_i. Let us consider the last $(i+1)$-th layer, whose input is a matrix of size $m_{i+1} \times n_{i+1}$. Clearly, the weight matrix of the $(i+1)$-th layer W_{i+1} contributing to the classification is the one $W_{i+1,i}$ of the $(i+1)$-th layer contributing to the i-th layer multiplied by the one W_i of the i-th layer. In detail, we first calculate each unit (*e.g.*, $W_{i+1}[r][c]$) in the input matrix of the $(i+1)$-th layer contributing to the i-th layer, yielding the intermediate weights (*e.g.*, $W^{r,c}_{i+1,i}$). This calculation depends on the layer structure. Then we multiply the intermediate weights by W_i and thus obtain W_{i+1}. Taking the unit $W_{i+1}[r][c]$ for example, we have $W_{i+1}[r][c] = \sum_{1 \le p \le m_i, 1 \le q \le n_i} W^{r,c}_{i+1,i}[p][q] \times W_i[p][q]$, where $1 \le r \le m_{i+1}$ and $1 \le c \le n_{i+1}$.

Opcode Filtering. After the weight matrix is calculated, we select the units of the top-k weights as a reference, which is used to filter the opcode sequences collected in Section II-C1. The opcode filtering is given in Algorithm 4, which takes an opcode sequence set and a set of keys (*i.e.*, the top-k units) as input, and returns another opcode sequence set. The algorithm iterates each opcode sequence as follows: it checks how many keys (*i.e.* 2-Gram sequences) are there appearing in the sequences and accumulate the corresponding weights; If there are some keys appearing in it (*i.e.*, $w > 0$), the sequence is kept in the result. Note that the accumulated weights of the selected sequences reflect how they contribute to the target family classification in some sense.

Algorithm 4 Opcode Filtering $OF(oSeqs, kKeys)$

```
Input: an opcode sequence set and a set of keys
Output: an opcode sequence set
 1: let $res = \emptyset$
 2: for $seq \in oSeqs$ do
 3:     let $w = 0$
 4:     for $2ops \in 2Gram(seq)$ do
 5:         if $2ops \in kKeys$ then
 6:             $w += 2ops.weight$
 7:         end if
 8:     end for
 9:     if $w > 0$ then
10:         $seq.weight = w$ and $res \cup = \{seq\}$
11:     end if
12: end for
13: return $res$
```

3) Behavior Analysis: A simple solution to mine the behavior for a target malware family is to compute the behaviors that only belong to the target family, that is,

$$Ch(F) = \bigcap_{apk \in F} seqs(apk) \setminus \bigcup_{apk \in F', F' \neq F} seqs(apk)$$

where F, F' are family names and $seqs$ is a function returning the selected opcode sequences presented in Section II-C2. However, this solution may yield no behaviors for some families. In this paper, we use vector space model (VSM) [20] to represent the malware families, that is, each malware family is represented as a vector as follows:

$$F_j = (r^j_1, \dots, r^j_n)$$

where F_j denotes the j-th family, n is the number of total selected behaviors for all families, and r^j_i is the relevance (or the importance) of the i-th behavior with respect to the j-th family. We characterize the relevances by the term frequency-inverse document frequency (TF-IDF), one of the popular statistical indicators used in the VSM:

$$r^j_i = tf^j_i \times idf^j_i$$

We define the term frequency tf^j_i as

$$tf^j_i = \frac{num(seq_i, F_j) * weight(seq_i, F_j)}{max_{i' \in [1,n]}(num(seq_{i'}, F_j) * weight(seq_{i'}, F_j))}$$

and the inverse document frequency idf^j_i as

$$idf^j_i = log \ \frac{m}{|\{F \mid seq_i \in seqs(F_j)\}| + 1}$$

where $num(seq_i, F_j)$ denotes the number of behavior seq_i appearing in the family F_j, $weight(seq_i, F_j)$ denotes the weight of behavior seq_i with respect to the family F_j calculated by Algorithm 4, and m is the number of malware families. The relevance r^j_i measures not only that the corresponding behavior seq_i appears quite often in the F_j family, but also that it is not frequent in the other families. As a result, it tends to filter out behaviors that are common across malware families. Finally, after all the vectors are calculated, we select the top-k behaviors as a characterization for the target family.

III. Experiment

This section presents our experimental results, including the family classification experimental results and the family characterization experimental results.

The malware samples we used are collected from Marvin [21], Drebin [11], VirusShare [22], and Contagio-Dump [23]: we first upload the samples[1] into VirusTotal and identity their labels as the most voted ones by those anti-virus tools gathered in VirusTotal via the tool *AVclass* [24]. As many families contain only few samples, we take the most common 20 families with 10675 malware, which are listed in Table I, wherein we take 90% of samples from each malware family as the training set and the remaining as the testing set.

[1]The samples from Drebin have been labelled, so we do not upload them.

TABLE I
MALWARE SAMPLES OF DIFFERENT FAMILIES

Id	Family	Num	Id	Family	Num
A	Adrd	134	K	GinMaster	1315
B	BaseBridge	673	L	GoldDream	90
C	DroidDream	90	M	Iconosys	180
D	DroidKungFu	1323	N	Imlog	57
E	ExploitLinuxLotoor	70	O	Kmin	1364
F	FakeDoc	131	P	MobileTx	130
G	FakeInstaller	925	Q	Opfake	2753
H	FakeRun	68	R	Plankton	811
I	Gappusin	176	S	SendPay	66
J	Geinimi	247	T	SMSreg	72

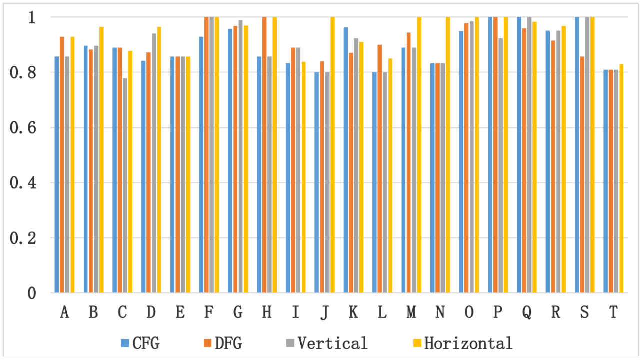


Fig. 2. Results on Different Features

A. Family Classification Experiments

In this section, we first conduct experiments to see the effectiveness of CFG and DFG in malware family classification. Then we present experiments to compare our approach with some related tools, namely, CDGDroid [15], Drebin [11], and VirusTotal [16].

1) Experiments on Different Features: We separately use the features CFG, DFG, and their combinations to build the classification model and conduct 10-fold cross validation experiments on the training set. The experimental results are given in Figure 2.

From the results, we can see that all the features are effective in classifying malware families, with the average *accuracy* 89.56%, 90.97%, 89.90% and 94.71%, respectively. Compared with CFG, DFG can classify more malware families with a better accuracy (10 betters for DFG while 5 betters for CFG). And considering the average accuracies, DFG performs a little better than CFG. This is mainly because that DFGs are built on CFGs such that DFGs would, in some sense, contain some "control flow" information.

Moreover, the results also show that the horizontal combination performs better than the vertical combination: (1) the horizontal combination performs better on more malware families with a better accuracy than the vertical combination; and (2) the average accuracy of the horizontal combination is higher than the one of the vertical combination. Rather surprisingly, the combination may make against the classification, for example, both combination modes perform a little worse than either CFG or DFG on Family C. But on average, both the combinations can offer a better performance for more families than either CFG or DFG does. Especially, the horizontal combination offers the best performance. A possible reason for this is that the horizontal combination can make full use of both CFG and DFG.

TABLE II
AVERAGE ACCURACY FOR OUR APPROACH AND CDGDROID

	CFG	DFG	Vertical	Horizontal
Our Approach	93.90%	92.35%	94.76%	95.46%
CDGDroid	87.24%	85.43%	86.56%	89.73%

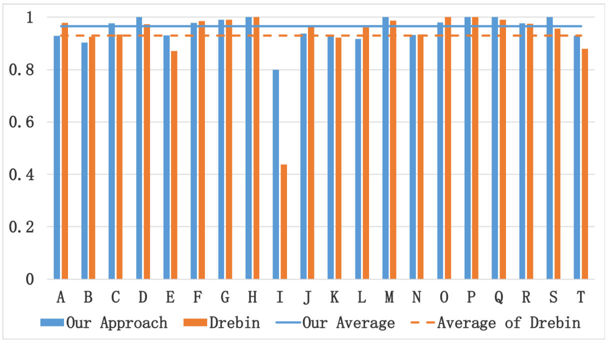


Fig. 3. Comparison against Drebin

2) Comparison Against Some Tools: In this section, we present experiments to compare our approach with CDGDroid, Drebin, and VirusTotal.

We first compare our approach against a natural extension of CDGDroid (from binary classification to multi-level classification) on the dataset. The comparison results are shown in Table II, where the average accuracy is counted without families. From the results we can see that our extended approach performs better than CDGDroid on all the features, in terms of average accuracy. The main reason is that our extended approach takes the permissions required by an invoke instruction into account.

Drebin is a lightweight Android malware detecting and classifying tool based on SVM, using 8 different types of features. We next conduct experiments to compare our approach against Drebin on the Drebin dataset. The experimental results are given in Figure 3, which show that (1) our approach performs better on more malware families than Drebin; and (2) the average accuracy of our approach is higher than the one of Drebin. The results also indicate that the features we consider (*i.e.* CFG and DFG) are quite effective in malware family classification, with respective to the 8 features used in Drebin.

Finally, we also compare our approach against some anti-virus tools gathered in VirusTotal. For that, we design a crawler to automatically upload the samples in the testing set collected from the Drebin dataset into VirusTotal for further detecting by those anti-virus tools. The results are shown in Table III, where those tools with too few responds from VirusTotal are filter out, the responds with a similar family

TABLE III
COMPARISON AGAINST ANTI-VIRUS TOOLS IN VIRUSTOTAL

Tool	A	B	C	D	E	F	G	H	I	J
our	90.000%	96.970%	88.889%	97.015%	85.714%	100.000%	96.774%	100.000%	83.333%	100.000%
Ad-Aware	–	93.939%	100.000%	35.821%	42.857%	100.000%	96.774%	85.714%	50.000%	100.000%
Arcabit	–	96.970%	100.000%	34.328%	28.571%	100.000%	95.699%	100.000%	83.333%	100.000%
Avast	100.000%	27.273%	33.333%	71.642%	42.857%	28.571%	51.613%	–	–	100.000%
AVG	100.000%	27.273%	66.667%	71.642%	42.857%	28.571%	51.613%	–	–	100.000%
Avira	40.000%	78.788%	77.778%	80.597%	–	35.714%	51.613%	–	–	100.000%
BitDefender	–	96.970%	100.000%	26.866%	100.000%	100.000%	94.624%	100.000%	–	100.000%
CAT-QuickHeal	–	100.000%	100.000%	85.075%	–	100.000%	77.419%	100.000%	100.000%	100.000%
ClamAV	–	75.758%	55.556%	82.090%	–	–	–	–	–	90.000%
DrWeb	10.000%	63.636%	33.333%	49.254%	–	92.857%	–	–	–	100.000%
Emsisoft	–	96.970%	100.000%	26.866%	42.857%	92.857%	98.925%	–	–	100.000%
ESET-NOD32	–	24.242%	100.000%	98.507%	71.429%	–	45.161%	–	16.667%	100.000%
F-Prot	10.000%	100.000%	100.000%	25.373%	71.429%	92.857%	88.172%	–	50.000%	100.000%
F-Secure	10.000%	100.000%	77.778%	91.045%	42.857%	100.000%	80.645%	–	33.333%	100.000%
Fortinet	10.000%	18.182%	100.000%	40.299%	71.429%	7.143%	87.097%	–	16.667%	100.000%
GData	–	75.758%	100.000%	92.537%	–	100.000%	94.624%	–	–	100.000%
Ikarus	–	57.576%	100.000%	62.687%	42.857%	21.429%	45.161%	71.429%	–	100.000%
Kaspersky	100.000%	100.000%	88.889%	89.552%	100.000%	100.000%	63.441%	100.000%	–	100.000%
Microsoft	–	27.273%	88.889%	70.149%	28.571%	–	11.828%	–	–	100.000%
NANO-Antivirus	–	96.970%	100.000%	25.373%	–	78.571%	70.968%	–	–	70.000%
Sophos	–	24.242%	66.667%	95.522%	–	–	26.882%	–	–	100.000%
Symantec	–	15.152%	33.333%	–	14.286%	–	–	–	–	80.000%
Tencent	50.000%	100.000%	66.667%	46.269%	85.714%	100.000%	97.849%	100.000%	16.667%	100.000%
TrendMicro-HouseCall	60.000%	93.939%	88.889%	77.612%	28.571%	71.429%	44.086%	14.286%	–	80.000%
Zillya	60.000%	27.273%	77.778%	85.075%	14.286%	57.143%	45.161%	–	33.333%	80.000%
ZoneAlarm	100.000%	100.000%	88.889%	7.463%	100.000%	100.000%	63.441%	100.000%	–	100.000%
AhnLab-V3	100.000%	9.091%	100.000%	100.000%	42.857%	100.000%	96.774%	100.000%	50.000%	100.000%
Antiy-AVL	90.000%	96.970%	88.889%	97.015%	71.429%	92.857%	97.849%	71.429%	–	100.000%
Baidu	–	27.273%	–	68.657%	14.286%	0.000%	30.108%	–	100.000%	100.000%
Kingsoft	–	–	44.444%	–	–	78.571%	55.914%	71.429%	33.333%	30.000%
VBA32	90.000%	100.000%	22.222%	83.582%	42.857%	64.286%	48.387%	–	–	80.000%

Tool	K	L	M	N	O	P	Q	R	S	T
our	91.176%	85.714%	100.000%	100.000%	100.000%	100.000%	98.387%	96.825%	100.000%	80.000%
Ad-Aware	76.471%	100.000%	81.250%	–	100.000%	100.000%	96.774%	79.365%	–	–
Arcabit	76.471%	100.000%	87.500%	–	93.333%	100.000%	88.710%	82.540%	–	20.000%
Avast	97.059%	–	25.000%	100.000%	100.000%	100.000%	17.742%	–	–	40.000%
AVG	97.059%	–	25.000%	100.000%	100.000%	100.000%	17.742%	–	–	40.000%
Avira	79.412%	42.857%	25.000%	20.000%	40.000%	100.000%	–	98.413%	–	60.000%
BitDefender	76.471%	100.000%	87.500%	–	93.333%	100.000%	–	84.127%	–	20.000%
CAT-QuickHeal	100.000%	100.000%	100.000%	–	73.333%	100.000%	–	95.238%	–	40.000%
ClamAV	–	42.857%		20.000%	100.000%	100.000%	–	–	–	–
DrWeb	–	100.000%	81.250%	100.000%	80.000%	28.571%	–	–	66.667%	–
Emsisoft	76.471%	100.000%	–	–	93.333%	100.000%	–	84.127%	–	40.000%
ESET-NOD32	100.000%	57.143%	–	80.000%	86.667%	100.000%	–	98.413%	–	40.000%
F-Prot	88.235%	100.000%	100.000%	100.000%	60.000%	100.000%	70.968%	92.063%	–	40.000%
F-Secure	100.000%	42.857%	93.750%	–	86.667%	85.714%	16.129%	3.175%	–	20.000%
Fortinet	29.412%	100.000%	12.500%	–	60.000%	–	17.742%	7.937%	–	–
GData	23.529%	100.000%	–	–	100.000%	85.714%	91.935%	84.127%	–	40.000%
Ikarus	88.235%	100.000%	87.500%	100.000%	93.333%	85.714%	3.226%	14.286%	–	20.000%
Kaspersky	100.000%	100.000%	100.000%	60.000%	66.667%	100.000%	80.645%	95.238%	100.000%	100.000%
Microsoft	32.353%	–	–	–	100.000%	–	6.452%	47.619%	–	–
NANO-Antivirus	70.588%	42.857%	100.000%	20.000%	66.667%	100.000%	72.581%	3.175%	100.000%	–
Sophos	79.412%	100.000%	93.750%	–	86.667%	100.000%	40.323%	4.762%	100.000%	–
Symantec	41.176%	28.571%	100.000%	–	–	71.429%	3.226%	–	–	–
Tencent	91.176%	100.000%	75.000%	100.000%	100.000%	71.429%	69.355%	–	100.000%	40.000%
TrendMicro-HouseCall	32.353%	28.571%	12.500%	–	60.000%	28.571%	51.613%	17.460%	16.667%	20.000%
Zillya	26.471%	57.143%	6.250%	100.000%	46.667%	–	–	39.683%	16.667%	–
ZoneAlarm	100.000%	100.000%	100.000%	60.000%	66.667%	100.000%	80.645%	93.651%	100.000%	100.000%
AhnLab-V3	100.000%	42.857%	100.000%	–	100.000%	100.000%	87.097%	14.286%	66.667%	20.000%
Antiy-AVL	100.000%	100.000%	100.000%	100.000%	86.667%	100.000%	98.387%	3.175%	100.000%	100.000%
Baidu	100.000%	100.000%	100.000%	100.000%	100.000%	–	–	100.000%	–	–
Kingsoft	61.765%	–	62.500%	20.000%	–	42.857%	64.516%	11.111%	16.667%	60.000%
VBA32	32.353%	100.000%	–	20.000%	40.000%	14.286%	54.839%	17.460%	83.333%	60.000%

name are conservatively considered correct[2], and – denotes that there are no correct responds in term of family name.

From the results, we can see that not all the family in the Drebin dataset are considered by all the tools, except the J family (*i.e.*, Geinimi): the numbers of malware families that the tools in VirusTotal can detect on this testing set (*i.e.*, the accuracy is nonzero) range from 8 to 19 and the numbers of tools by which each malware family can be detected are from 12 to 30. This is mainly because different tools take different family classification systems. Moreover, the results also show that our approach performs better than 73.333% tools (or does not perform worse than all the tools) on more than half the malware families that the tool can detect on this testing set. And on no malware families there are 6 tools performing better than ours. In particular, the tool Kingsoft perform worse than our approach on all the malware families that Kingsoft can detect. Besides, for each family, except the L family (*i.e.*, Glodream), most tools that can detect the target family (*i.e.*, tools with a nonzero accuracy) have a lower accuracy than our approach on the testing dataset. In particular, there are 9 families for which no tools perform better than our approach on the testing dataset.

[2]There may be two different names that characterize an identity family, which are ignored here.

B. *Family Characterization Experiments*

In this section, we present the experimental results about the family characterization on some selected malware families.

1) Family Characterization: Due to space limitation, we present the top-20 behaviors extracted from CFG and DFG for Family R (*i.e.*, Plankton), which are given in Table IV, where we use 5-gram sequences to represent a behavior and select the 2-gram sequences of top-100 weight as a reference to filter

TABLE IV
TOP-20 BEHAVIORS FOR FAMILY PLANKTON

Graph	5-Gram
CFG	move-object/from16,move-exception,goto/16,move-object/from16,move-exception
	move-result-object,move-exception,monitor-exit,throw,move-exception
	const/16,const/16,aput-byte,const/16,const/16
	const-string,move-exception,monitor-exit,throw,move-exception
	move-object/from16,move-exception,invoke-static,new-instance,invoke-direct
	aput-byte,const/16,const/16,aput-byte,const/16
	monitor-exit,move-exception,monitor-exit,throw,move-exception
	const/16,aput-byte,const/16,const/16,aput-byte
	const-string,sput-object,const-string,move-exception,invoke-virtual
	move-result-object,move-exception,monitor-exit,move-exception,monitor-exit
	sget-object,move-exception,monitor-exit,throw,move-exception
	iget-object,move-exception,monitor-exit,throw,move-exception
	invoke-static,move-exception,monitor-exit,throw,move-exception
	move-object/from16,move-exception,const-string,move-object/from16,invoke-static
	move-object/from16,move-exception,invoke-direct/range,const-string,new-instance
	move-object/from16,move-exception,sget-object,const-wide/32,invoke-static/range
	move-object/from16,move-exception,sget-object,const-wide/32,move-exception
	move-object/from16,move-exception,sget-object,move-exception,invoke-static
	move-object/from16,move-exception,invoke-direct/range,const-string,move-object/from16
	move-object/from16,iget-object,move-object/from16,move-object/from16,move-exception
DFG	move/from16,invoke-virtual,move-result,move/from16,invoke-virtual
	move-object/from16,invoke-virtual,move-result,move/from16,invoke-virtual
	move-result-object,invoke-interface,move-result-object,invoke-interface,move-result-object
	move-result-object,invoke-static,move-result-object,invoke-static,move-result-object
	invoke-virtual,move-result-object,invoke-interface,move-result-object,invoke-interface
	move-result,iget-object,iget-object,iget-object,iget-object
	move-object/from16,iget-object,iget-object,move-object,move-object
	invoke-interface,move-result-object,invoke-interface,move-result-object,invoke-interface
	invoke-static,move-result-object,invoke-static,move-result-object,invoke-static
	move-result-object,invoke-interface,move-result-object,invoke-interface,move-result
	move/from16,invoke-virtual,move-result,move/from16,android.permission.DUMP
	move-object/from16,iget-object,move,move,invoke-virtual
	move-object/from16,iget-object,iget-object,iget-object,move-object
	move-object/from16,invoke-virtual,move-result,move/from16,android.permission.DUMP
	iput-object,iget-object,iput-object,iget-object,throw
	invoke-interface,move-result-object,invoke-interface,move-result,if-eqz
	move-object/from16,iget-object,move,move,move
	const/4,move,add-int/lit8,invoke-virtual,move-result
	move-object/from16,iget-object,iget-object,move-object,iget-objectl
	move-object/from16,iget-object,iget-object,move-object,invoke-virtual

behaviors, and the final total number of the selected behaviors for CFG and DFG are 1430127 and 55488, respectively.

From the table, we can see that the most possible common behaviors that shared by the samples of the Plankton family from CFG and DFG are "move-object, move-exception, goto, move-object, move-exception" and "move, invoke-virtual, move-result, move, invoke-virtual", respectively. In detail, all the top-20 behaviors from CFG are related to data manipulation, wherein 16 behaviors (80%) involve at least two data manipulations, and there are 8 behaviors (40%) involving function invokes. While for DFG, there are 19 behaviors (95%) that are related to data manipulation, wherein 18 behaviors (90%) involve at least two data manipulations, and 14 (70%) involving function invokes. This is in accordance with the behaviors of the Plankton family: most Plankton samples is included in host apps by adding a background service, which collect information, including the device ID as well as the list of granted permissions to the infected app, and send them back to a remote server. Moreover, there are 2 behaviors from DFG that request the DUMP permission, which allows an APK to get the dump system information from the System Services; and there are 17 behaviors from CFG that involve exceptions and may deserve attention.

2) Evaluation on Characterization: To evaluate the characterization, we build a classification model with top-k behaviors as features and then perform the model on the testing dataset. Figure 4 shows the experimental results about the six largest malware families with top-1‰ behaviors.

In general, for both CFG and DFG, the more behaviors, the higher accuracy. Specifically, the accuracy increases rapidly when we select the top-2‱ behaviors for CFG and the top-7‱ for DFG, respectively; while it increases slowly after that. This indicates that the characterization can capture the behaviors of the testing families.

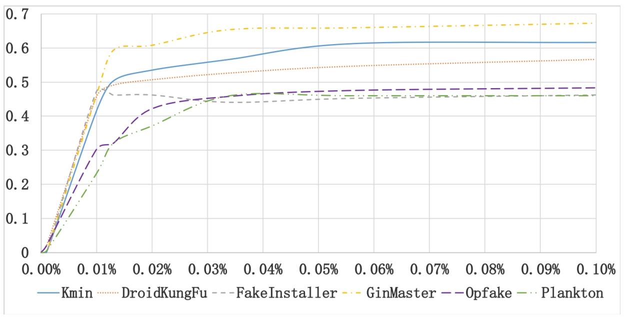
IV. RELATED WORK

Over the past decade, there are a lot of research work for Android malware analysis. Here we only review some related and recent ones, namely, malware classification and malware characterization.

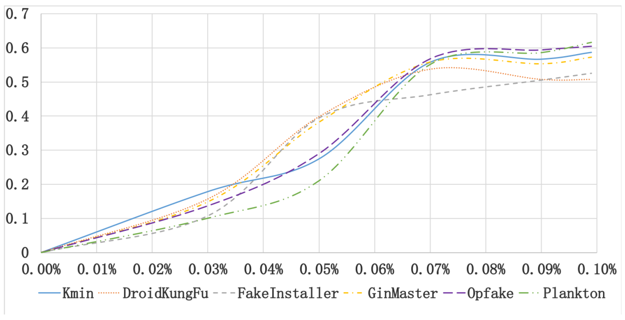
DroidAPIMiner [25] extracts critical API calls, their package level information, as well as their parameters as features to characterize Android malware. Hyunjae Kang et. al. [26] proposed an Android malware detection and classification system based on static analysis by using serial number information from the certificate as a feature. Richard Killam et. al. [12] proposed a system for classifying Android malware by leveraging the text strings present in an APK's binary file. Drebin [11] 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, to classify Android malware. Hein and Myo [13] proposed a characterization of Android malware, where permission, API calls and strings are used as features. Martín et. al. [14] proposed another android malware characterization using metadata, such as permissions, the application developer and certificate issuer. Nannan XIE et. al. [27] proposed a framework to explore the key features for Android malware families from three categories of features, that is, platform-based permissions, hardware components, and suspicious API calls. However, most of these approaches consider neither control flow properties, nor data flow properties.

Dendroid [28] is a text mining approach to analyzing and classifying android malware families, where the feature they used are a high-level representation of classic control-flow graphs. DroidMiner [7] uses a two-level behavioral graph, which is built on control-flow graphs and call graphs, to represent APKs, and then identifies and labels elements of the graph that capture malicious behavioral patterns. 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 family classification. DroidADDMiner [8] uses features based on data dependency between sensitive APIs to detect, classify and characterize Android malware. ASTROID [9] tries to look for a maximally suspicious common subgraph, which is built from inter-component call relations and their semantic metadata (*e.g.*, data-flow properties), such that the subgraph is shared between all known instances of a malware family, using Maximum Satisfiability. EnMobile [10] is a new entity-based characterization of Android application behaviors, which includes four types of predicates: event predicate, entity predicate, data-flow predicate, and control-flow predicate. All these tools can characterize malware behaviors on the high level semantically. But different from these work, our approach work on the instruction level and use the lightweight analysis, and thus is simpler.



(a) Model with Top-k Behaviors from CFG



(b) Model with Top-k Behaviors from DFG

Fig. 4. Accuracy for Model with Top-1‰ Behaviors

V. Conclusion

In this work, we have proposed an Android malware family classification and characterization approach, using control flow graph (CFG) and data flow graph (DFG) as features. To evaluate the proposed approach, we have carried out some interesting experiments. Through experiments, we have found that the family classification model taking the horizontal combination of CFG and DFG as features performs the best and our family classification model has a better performance than CDGDroid, Drebin and most of anti-virus tools gathered in VirusTotal. The experimental results have also shown that our characterization can capture the behaviors of the testing families.

As for future work, we may consider the key instructions to improve the approach. We can use other program graphs, such as program dependence graphs, to train the model. We can also consider the corresponding instructions of the top-k behaviors to explain the target malware families. More experiments on malware anti-detecting techniques (*i.e.*, obfuscation techniques) are under consideration.

Acknowledgements

This work was partially supported by the National Natural Science Foundation of China under Grants No. 61772347, 61836005, 61532019 and 61761136011, 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] *IDC Report*, http://www.idc.com/promo/smartphone-market-share/os.
[2] *Report from G DATA*, 2017, https://www.gdatasoftware.com/blog/2018/11/31255-cyber-attacks-on-android-devices-on-the-rise.
[3] J. Sahs and L. Khan, "A machine learning approach to android malware detection," in *EISIC '12*, 2012, pp. 141–147.
[4] K. Allix and et al., "Empirical assessment of machine learning-based malware detectors for android," *Empirical Software Engineering*, vol. 21, no. 1, pp. 183–211, 2016.
[5] N. A and et al., "Adaptive and scalable android malware detection through online learning," in *IJCNN '16*, 2016, pp. 157–175.
[6] N. Mclaughlin and et al., "Deep android malware detection," in *CODASPY '17*, 2017, pp. 301–308.
[7] C. Yang and et al., "DroidMiner: Automated Mining and Characterization of Fine-grained Malicious Behaviors in Android Applications," in *ESORICS '14*, 2014, pp. 163–182.
[8] Y. Li and et al., "Detection, Classification and Characterization of Android Malware Using API Data Dependency," in *SecureComm '15*, 2015, pp. 23–40.
[9] Y. Feng and et al., "Automated synthesis of semantic malware signatures using maximum satisfiability," in *NDSS*, 2017.
[10] W. Yang, M. R. Prasad, and T. Xie, "Enmobile: Entity-based characterization and analysis of mobile malware," in *ICSE*, 2018.
[11] D. Arp and et al., "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket," in *NDSS '14*, 2014.
[12] P. C. R. Killam and N. Stakhanova, "Android malware classification through analysis of string literals," in *Workshop on TA-COS*, 2016.
[13] C. L. P. M. Hein and K. M. Myo, "Characterization of malware detection on android application," in *Genetic and Evolutionary Computing*, 2016, pp. 113–124.
[14] I. Martín and et al., "Android malware characterization using metadata and machine learning techniques," *CoRR*, vol. abs/1712.04402, 2018.
[15] Z. Xu and et al., "CDGDroid: Android Malware Detection Based on Deep Learning Using CFG and DFG," in *ICFEM'18*, 2018, pp. 177–193.
[16] *VirusTotal*, https://www.virustotal.com.
[17] K. W. Y. Au and et al., "Pscout: Analyzing the android permission specification," in *CCS*, 2012.
[18] *Dalvik Bytecode*, https://source.android.com/devices/tech/dalvik/dalvik-bytecode.
[19] *Android Permission Overview*, https://developer.android.com/guide/topics/permissions/overview.
[20] G. M. Salton, A. Wong, and C. S. A. Yang, "A vector space model for automatic indexing," *Communications of the Acm*, vol. 18, no. 11, pp. 613–620, 1974.
[21] M. Lindorfer, M. Neugschwandtner, and C. Platzer, "Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis," in *ComSAC '15*, 2015, pp. 422–433.
[22] *VirusShare*, https://virusshare.com/.
[23] *Contagiodump*, http://contagiodump.blogspot.com/.
[24] M. Sebastián and et al., "AVclass: A Tool for Massive Malware Labeling," in *International Symposium on Research in Attacks*, 2016.
[25] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *SecureComm*, 2013.
[26] H. Kang and et al., "Detecting and classifying android malware using static analysis along with creator information," *International Journal of Distributed Sensor Networks*, vol. 11, no. 6, pp. 1–9, 2015.
[27] N. Xie and et al., "Fingerprinting android malware families," *Frontiers of Computer Science*, no. 12, pp. 1–10, 2018.
[28] G. Suarez-Tangil and et al., "Dendroid: A text mining approach to analyzing and classifying code structures in android malware families," *Expert Systems with Applications*, vol. 41, no. 4, Part 1, pp. 1104 – 1117, 2014.