How I met your mother?
An empirical study about Android Malware Phylogenesis

Gerardo Canfora[1], Francesco Mercaldo[1,2], Antonio Pirozzi[1] and Corrado Aaron Visaggio[1]
[1]*Department of Engineering, University of Sannio, Benevento, Italy*
[2]*Centro Regionale Information Communication Technology - CeRICT scrl, Benevento, Italy*
{*canfora, fmercaldo, pirozzi, visaggio*}*@unisannio.it*

Keywords: malware, phylogenesis, lineage, triage, security, Android

Abstract: New malware is often not really new: malware writers are used to add functionality to existing malware, or merge different pieces of existing malware code. This determines a proliferation of variants of the same malware, that are logically grouped in "malware families". To be able to recognize the malware family a malware belongs to is useful for malware analysis, fast infection response, and quick incident resolution. In this paper we introduce DescentDroid, a tool that traces back the malware descendant family. We experiment our technique with a real world dataset of malicious applications labelled with the family they belong to, obtaining high precision in recognizing the malware family membership.

1 INTRODUCTION

In the recent years Android platform has increasingly been targeted by malware (F-Secure, 2015). This high volume of new malware is not really "new". As a matter of fact malware, like any software, evolves. Malware writers modify existing code in ways that are typical in software industry and open source landscape (Walenstein and Lakhotia, 2012): they add features to an existing malware, they can generate multiple configurations of the same malware to allow the execution on several platforms, they merge together components of different malware programs. Other methods for changing malware include: recompiling (Rosenblum et al., 2011), packing (Chen et al., 2008), permuting, obfuscating (Nagra and Collberg, 2009), or otherwise tweaking programs (Schipka, 2007).

The study of the malware evolution can concern four different classes of problems: phylogenetic analysis, lineage reconstruction, variants identification, and code clones search. Phylogenetic analysis is the study of similarities and differences in program structure to find relationships within groups of software programs, providing insights about new malware variants not available within the databases of malware signatures (Jilcott, 2015). Lineage reconstruction is the identification of the ancestor-descendant relationships among malware samples, identifying, if possible, the direct samples from which a specific piece of malware may have been derived (Dumitras and Neamtiu, 2011).

Variants identification consists of localizing malware samples that introduce an evolution with respect to an existing malware (Jang et al., 2011). Finally, finding code clones aims to retrieve which pieces of codes are reused within a new malware sample (Farhadi et al., 2014). Recognizing the relationships among malware programs is at the basis of a variety of security tasks, from malware characterization to threat detection and cyber-attack prevention. In malware triage (Hu et al., 2009; Bayer et al., 2009; Jang et al., 2011; Battista et al., 2016), lineage can be used by malware analysts to understand trends over time and make informed decisions about the best strategies to dissect the malware samples. Moreover, identifying malware lineage and phylogenetic can help to face more promptly zero day malware programs, when they are or contain evolved versions of known malware. The main limit of the methods already proposed in literature is that they extract features at a single level of program abstraction; thus, malware writer could make ineffective the detection technique by adopting an obfuscation technique at that level of abstraction.

In this paper we introduce a method able to assign a malware variant to the family it belongs to by using three different levels of program abstraction. The three levels have been chosen to contrast the different types of obfuscation that may be adopted by malware writers at the different layers of a program. We

consider these levels as: op-code, Control Flow and Function Call Graphs (CFG and DCG), and the sequence of system calls produced by the execution of the malware. These features are then used for computing two different functions of similarity, that are considered as the metrics for the classification phase. In our evaluation, the method showed a very high precision in recognizing the provenance family of a malware.

The paper proceeds as follows. Section 2 introduces the related literature divided into two research topics: the building of malware phylogenesys, and the identification of malware variants; Section 3 describes the method and Section 4 discusses the evaluation of the method; finally, Section 5 draws the conclusions.

2 RELATED WORK

The problem of recognizing the malware families mainly refers to two areas of research in malware analysis: the building of malware phylogenesys, and the identification of malware variants. Goldberg et al. (Goldberg et al., 1998) studied malware phylogeny by applying suffix trees to build phyloDAGs.

Erdelyi and Carrera (Carrera and Erdélyi, 2004) through phylogenetic trees computed the relationships between 6 distinct groups of malware extracting the function call graphs. Karim et al. (Karim et al., 2005) used n-perms along with n-grams on op-code sequences to classify malware and to compare with existing classification schemes. Ma et al. (Ma et al., 2006) studied the diversity of shellcode by computing exedit distances of instruction byte sequences produced with code emulation. Wehner (Wehner, 2007) showed how families of internet worms were related by their normalized compression distance, leveraging phylogenetic trees. Kong and Yan (Kong and Yan, 2014) introduced the transductive malware classification, a paradigm to infer family information of malware variants under certain circumstances.

Giannella and Bloedorn (Giannella and Bloedorn, 2015) developed a version of spectral clustering applied to behavior-based malware clustering, which was a technique proposed and investigated by Rieck et al.(Rieck et al., 2011). The main limit of this method stands in the computational cost, which is high with respect to baseline algorithms, as stated by the authors. Our method outperforms this method in terms of precision. Zhong et al. (Zhong et al., 2012) proposed a classification method based on function level similarity comparison. The main difference with our work is that Zhong et al. take into account only metamorphic malware, while our method is thought to analyze any kind of malware. Zhong et al. (Zhong et al., 2013) proposed a system that classifies malware by extracting a set of code metrics from the programs and computing the distance between programs. The authors use a dataset that is much smaller than the one used in our experiment and they measured performances that are lower than ours. Yu and colleagues (Yu et al., 2010) proposed a byte frequency based detecting model to identify malware variants. Shang et al. (Shang et al., 2010) presented a method aimed to identify win32 malware variants computing similarity between two binaries on the basis of their function callgraph similarity: authors conclude that the method has many drawbacks to fully realize malware automatic analysis. The approach proposed by Cesare et al. (Cesare and Xiang, 2011) builds a birthmark of a malware based on the set of control graphs it has. Authors conclude that using 10 000 samples the false positive rate is less than 1%. Also Wu et al. (Wu et al., 2013) based their work on function-call graphs in order to identify metamorphic malware. They evaluate their technique using more then 200 pairs of malware vairiants obtaining a similarity score ranging from 0.2 to 0.4. Agrawal et al. (Agrawal et al., 2012) proposed an abstract malware signature based on extracting semantic summaries of malware code, obtaining a true positive rate equal to 86%. Xiaofang et al. (Xiaofang et al., 2014) proposed a similarity search of malware variants using distance metrics based on locality-sensitive hashing schemes, obtaining a precision equal to 0.9. Shen et al. (Shen et al., 2014) proposed a technique consisting in detecting malware variants generated by various obfuscation techniques, by using the topology graph of payloads in order to model relationships between components and the API sets. They obtained a detection ratio ranging from 78.26% to 100.0%. Azab et al. (Azab et al., 2014) investigated whether the Trend Locality Sensitive Hashing algorithm is useful to group binaries that belong to the same variant, using the k-NN algorithm. They obtain an accuracy equal to 0.989 using 878 binaries. As it emerges from this discussion and at the best knowledge of the authors, the method presented in this paper is new, because it uses three different program abstractions and similarity functions never applied before.

3 THE METHOD

The main goal of *DescentDroid* is to identify the malware family a given malware has the highest probability to belong to. Our model is made of 4 stages organized as a pipeline, where each stage generates the

input for the next one:

1. relative op-code frequency distribution analysis;
2. CFG/FCG extraction;
3. isomorphism analysis with *n*grams;
4. classification.

Each malware program m is associated to a vector of features v_m extracted from the code at three different levels of program abstraction: op-code listings, Control Flow Graph and Function Call Graph, and sequences of syscalls.

The first feature we extract is the *op-code frequency distribution*: previous works (Canfora et al., 2015), (Canfora et al., 2016), (Mercaldo et al., 2016) demonstrated that this feature is able to effectively identify a malware family. Op-code frequency distribution can be ineffective when obfuscation techniques are adopted. Because of this consideration, we extract a second class of features, which characterize the CFG and the FCG of a malware application. Some papers (Kinable and Kostakis, 2011) and (Gascon et al., 2013) show that it is possible to accurately detect malware families via call graph clustering and function call graphs, and such techniques are robust against obfuscation. In fact, it is observed that the Control Flow is invariant between different mutations of a worm (Kruegel et al., 2005).

The third class of features comprehends sequences of system calls. These features are then used for computing two different types of distances, that will be used to establish the similarity between malware program m and a malware Family F. This similarity is obtained by the Isomorphism Analysis on the *n*gram extracted both by the CFG and by the FCG.

The last stage is the classification of the malware program m_x, accomplished through a similarity evaluation between a malware m_x and a candidate family $\mathrm{F_i}$, computed with two similarity scores, namely: $\mathrm{SC_1(m_x, F_i)}$, and $\mathrm{SC_2(m_x, F_i)}$.

These scores are functions of the three measures extracted with the previous steps, described above.

A malware program m_x is assigned to a malware family F_i if and only if the values of both the scores for the malware compared with the correspondent values of scores for that family are smaller than the fixed threshold. Given $\mathrm{t_1}$ the threshold for the similarity score $\mathrm{SC_1}$, and $\mathrm{t_2}$ the threshold for the similarity score $\mathrm{SC_2}$, $\mathrm{m_x} \in F_\mathrm{i} \iff SC_1(m_\mathrm{x}, F_\mathrm{i}) < t_1 \bigwedge SC_2(m_\mathrm{x}, F_\mathrm{i}) < t_2$.

The thresholds have been established by empirical trials with 1000 samples belonging to the training dataset, choosing the values which maximize the accuracy: 0.0002 for $\mathrm{SC_1}$, and 0.10003 for $\mathrm{SC_2}$.

In the following subsections we describe each stage in detail.

3.1 Stage 1: Relative Op-codes Frequency Distribution Analysis

In this phase we extract the *Relative Op-codes Frequency Distribution*. The business logic of an Android application is contained in the .dex file used by dalvik, an implementation of Java virtual machine for Android environment. We extract the relative frequency of the op-codes from the dex file: there are 255 different op-codes in the *Official Dalvik Bytecode Set Table* (Dalvik, 2015). Op-codes implementing similar functions are grouped in the same category, e.g. op-code related to move operations (i.e., move, move/from16, move/16, move-wide, move-wide/from16, move-wide/16, move-result, move-result-wide, move-result-object op-code) are grouped in *move* category.

Once extracted, the relative frequency of each op-code is stored in a fixed-size vector with 29 elements, where each element represents the op-code category (a density measure) the corresponding op-code falls in. Then, we compare the vector extracted from m_x with the vectors representing the families of the training dataset by using cosine similarity, in order to obtain the 10 closest vectors (families) to m_x. The Cosine Similarity, is then computed for each of the Dalvik Op-codes Category. The closer to 1 the distance is, the more similar two vectors are.

3.2 Stage 2: CFG/FCG Extraction

We extract the CFG and the FCG from the applications with AndroGuard (Androguard, 2015). Both the CFG and the FCG of each candidate malware program m_x is extracted in the form of Adjacency Lists. These Adjacency Lists are compared with the Adjacency Lists of the CFG and FCG of malware samples contained in the collection of the training set through a Similarity function for identifying common shared structures between the candidate malware program m_x and the malware families F_i. Finally, these Adjacency Lists are ready for the Isomorphism analysis stage with the n-grams analysis.

3.3 Stage 3: Isomorphism analysis with n-grams

In this phase, we detect the contiguous sequence of the instructions (CFG) or the contiguous sequence of the invoked syscalls (FCG) that are in common between the candidate malware program m_x and the

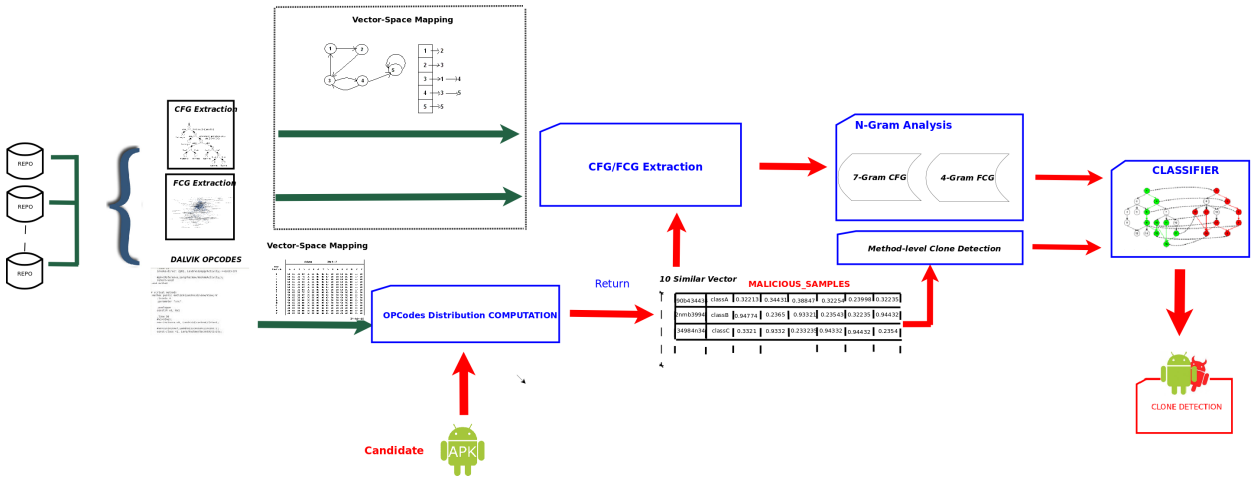

Figure 1: the 4-staged Architecture of *DescentDroid*

malware families F_i. This operation is equivalent to extract common (CFG and FCG) subgraphs between the candidate malware program m_x and the malware families F_i. We consider 7-grams regarding the CFG Adjacency Lists and 4-grams with regards to the FCG Adjacency Lists. The choice of the *n*grams parameter in the CFG and FCG is driven by an empirical analysis i.e., we noticed that a smaller sequence is likely to produce a high number of false positives. For this reason we chose the smallest n that allows to consider only the *relevant* portions of the graph. The *n*grams analysis detects the common maximum subgraph of the candidate with the 10 closest malware samples returned by the op-code Frequency Distribution Analysis. In addition to the *n*grams analysis, phase 3 performs the method-level similarity analysis for the identification of code clones at method-level. This analysis compares the op-code Frequency Distribution for each method with the distribution of the top 10 returned vectors. The choice of comparing the op-code distribution instead of the methods name is driven by the fact that, in a code-reuse scenario, methods name or more in general *identifiers* can be changed or encrypted, while an identical op-codes distribution still remains indicative of a code clone. This analysis is resilient at least for type I, and type II Clones but not for type III and type IV, in fact, type I and II still have the same op-codes distribution (Roy et al., 2009).

3.4 Stage 4: Similarity Scores computation

The last phase computes the similarity score which evaluates the affinity of the given malware m_x to one or more malware families F_i through the classification. The malware variants detection is based on a threshold analysis based on isomorphism ratio between malware samples.

The similarity scores we define take into account also the numbers of n-grams detected in the Call Graphs: if there are not *n*grams taken from CFG neither from the FCG, then the Similarity Score is 0. This means that there is no similarity between the candidate m_x and the malware families F_i of the training set. The classification is accomplished through two Similarity Scores. $\mathbf{SC_1}$ is defined as:

$$max(\#7-gramsCFG, \#4-gramsFCG) * methodSimRatio \quad (1)$$

In this formula, we assume that shared n-grams for CFG or FCG exist: we take the maximum value of the common n-grams (7-grams for the Control Flow Graphs, and 4-grams for the Function Call Graphs) and then multiply it with the *methodSimRatio* variable. The *methodSimRatio* is a variable that spans in the interval [0,1] and is defined as follows:

$$methodSimRatio = \frac{countOfClonedMethods}{totalMethods} \quad (2)$$

methodSimRatio is the ratio of cloned methods between the candidate apk and the *i*-th apk in the training dataset. 0 means that there are not cloned methods whereas 1 means that all the methods between the two apps were cloned. Then we also defined the OpCodeSimScore spanning in the interval [0,1] as a variable that expresses the similarity between the Frequency Distribution of the Dalvik Op-codes of the 2 apps. The *OpCodeSimScore* is a variable defined as follows:

$$OpCodeSimScore = cos(\theta) = \frac{M_{iC} \cdot F_{iC}}{\|M_{iC}\| \|F_{iC}\|} \quad (3)$$

where M_{iC} is the Relative Frequency of that *i-th* Category of Op-code for the candidate malware m_x and F_{iC} is the Relative Frequency of that *i-th* Category of Op-code for the Family F_i that is part of the training dataset. In this formula we don't take into account the *Relative op-codes Frequency Distribution Sim Score* because, the 10 returned vectors are yet sorted by this score. $\mathbf{SC_2}$ is defined as:

$$OpCodeSimScore * (\#7 - gramsCFG) * (\#4 - gramsFCG) * methodSimRatio \quad (4)$$

Both the similarity scores are normalized so that when a similarity score is close to 1 it means that the similarity is maximum (1 means that the app is a perfect clone), while 0 means that there is not similarity at all.

4 EVALUATION

The aim of the experiment is to evaluate the effectiveness of *DescentDroid* in identifying the family a malware program belongs to. The method consists of classifying malware by using the threshold based mechanism described previously. The classification results are measured with three metrics : precision, recall, and accuracy. The precision has been computed as the proportion of the examples that truly belong to class X among all those which were assigned to the class:

$$Precision = \frac{tp}{tp + fp} \quad (5)$$

where *tp* indicates the number of true positives and *fp* indicates the number of false positives. The recall has been computed as the proportion of examples that were assigned to class X, among all the examples that truly belong to the class, i.e. how much part of the class was captured:

$$Recall = \frac{tp}{tp + fn} \quad (6)$$

where *fn* is the number of false negatives. Then we compute the Accuracy defined as the overall correctness of the model, computed as the sum of correct classifications divided by the total number of classifications:

$$Accuracy = \frac{tp + tn}{tp + fp + tn + fn} \quad (7)$$

4.1 The Dataset

We built a dataset composed of 4000 Android malware applications of different kinds and covering a wide spectrum of malicious intents from Drebin Dataset (Arp et al., 2014; Spreitzenbarth et al., 2013). Malware dataset is also partitioned according to the *malware families*: each family contains samples which have in common several characteristics, like payload installation, the kind of attack and events that trigger malicious payload (Zhou and Jiang, 2012). We used 3000 malware samples as training dataset, while the remaining 1000 have been used to test the effectiveness of the proposed technique.

4.2 Analysis of Results

We submitted to DescentDroid 1000 malware samples labelled with the correspondent provenance family: DescentDroid has correctly classified the family of 979 malware samples, and then we obtained an accuracy of:

$$Accuracy = \frac{979}{1000} = 97.9\% \quad (8)$$

The classification is done with the two features, namely the Final Similarity Scores 1 and 4 at the final stage. Table 1 shows the evaluation results, where each family is associated to the number of the samples it includes, and the precision and recall that we obtained with the classification. The malware submitted belonged to 114 different families, and *DescentDroid* identified the malware family with a precision ranging in most cases from 0.9 to 1. Precisely, all the malware samples belonging to 82 families were classified in the right family (i.e., with precision equal to 1) which corresponds to 57% of cases. This supports the effectiveness of *DescentDroid* to trace the provenance family of a malware. Only in the 9.5 % of cases, we obtain a precision equal to or smaller than 0.5.

We noticed that several families in the test set are represented by a few malware samples, thus we wonder if the classification performances is somehow related with the family size. For answering this question, we plotted in figure 2 and 3 the precision and the recall obtained from each family with the corresponding size of the family set: each circle represents a malware family and the radius of the circle is proportional to the number of samples for that family. About Precision, in Fig. 2 we can observe that *DescentDroid* achieves a high precision both for malware families with a few (little circles) or many (big circles) samples in the training dataset, except for 10 families: for those we measured a value of Precision under or near 0.5, as stated in Fig. 2. For all those samples that belong to these families, *DescentDroid* would not behave better than a random classifier. These results mean that the final Precision of the Classification process does not depend on the number of samples in the training set, but, could depend

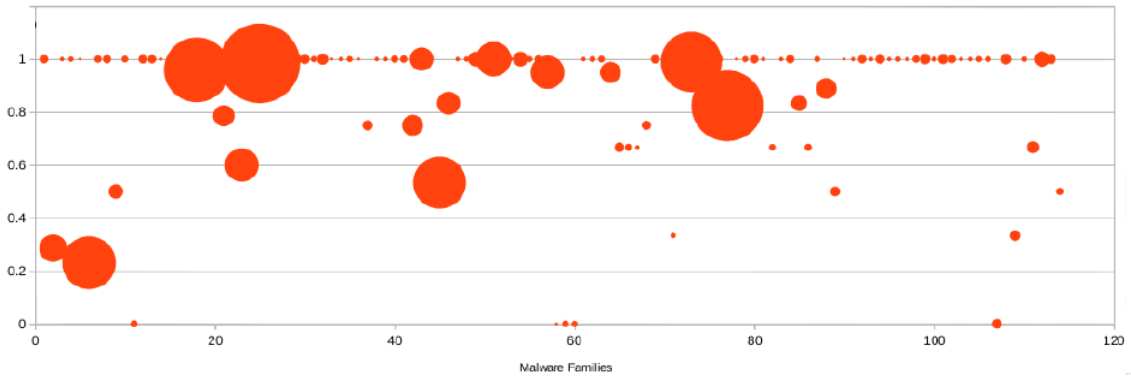

Figure 2: Precision obtained by the evaluation

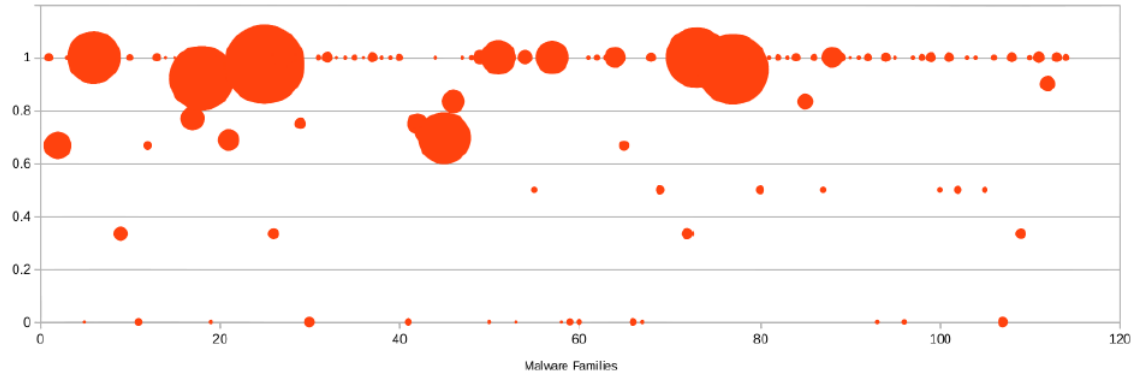

Figure 3: Recall obtained by the evaluation

on the level dissimilarity among the samples of the same family. For strengthening this conclusion we run a correlation analysis between the families size and the precision obtained, resulting a correlation index of 0.0658, which corresponds to a very irrelevant correlation, and thus confirms our conclusion. This may be a reasonable explanation, since some families count samples that are strongly different among each others and share a very small part of code or behavior. In this case *DescentDroid* lacks its effectiveness, as our heuristic is based on the measure of the isomorphism ratio between malware samples. Then we can deduce that, if the isomorphism ratio of a given malware family is low, therefore, the Precision for that family is low. For the evaluation we used a Apple Mac Pro 5.1, equipped with a Xeon QUADCore 2.66 GHz, 4GB RAM, and 320GB 7200 rpm HDD. The Execution Time for the whole analysis of a single malware sample depends strongly on the size of the sample. For the execution time, we obtained 0.33 min as the minimum, 2.15 min as the average, and 5.33 min as the maximum.

5 CONCLUSION AND FUTURE WORK

In this paper we present a method for deriving the family a malware belongs to. To achieve this goal, we apply a four staged process which extracts three different types of program abstractions, namely the op-codes Frequency distribution, CFG/FCG extraction and the sequence of system calls produced by the execution of the malware. Thus two similarities metrics are computed and used for accomplishing a threshold based classification. The method demonstrated its effectiveness as the 57% of families are recognized with precision equals to 1. As future work, we are improving our method for being effective also when a family shows a low rate of code and behaviors shared among samples, and, we are developing a method to characterize the multi-family derivation of a malware.

REFERENCES

Agrawal, H., Bahler, L., Micallef, J., Snyder, S., and Virodov, A. (2012). Detection of global, metamorphic

Table 1: Precision and Recall obtained with each family, with the number of samples tested for each family (between parentheses)

Family	P	R	Family	P	R
AccuTrack (2)	1	1	Adrd (3)	0.28	0.66
Adsms (1)	1	1	Aks (2)	1	1
Anti (1)	1	0	BaseBridge (3)	0.23	1
BeanBot (2)	1	1	Biige (2)	1	1
Boxer (3)	0.5	0.33	Ceshark (2)	1	1
Coogos (2)	0	0	Copycat (3)	1	0.67
Cosha (3)	1	1	Dabom (1)	1	1
Dialer (1)	1	1	Dougalek (3)	1	1
DroidDream (13)	0.9	0.77	Koomer (1)	0	0
DroidRooter (2)	0.96	0	DroidSheep (4)	1	1
ExploitLinuxL (16)	0.78	0.69	FaceNiff (1)	1	1
FakeDoc (3)	0.6	1	FakeFlash (1)	1	1
FakeInstaller (191)	0.98	0.97	FakePlayer (3)	1	0.33
FakeRun (18)	1	1	FakeTimer (3)	1	1
Fakelogo (4)	1	0.75	Fakengry (2)	1	0
FarMap (1)	1	1	Fatakr (3)	1	1
Fauxcopy (1)	1	1	FinSpy (1)	1	1
Fjcon (2)	1	1	Flexispy (1)	1	1
FoCobers (3)	0.75	1	Fsm (1)	1	1
GPSpy (1)	1	1	Gamex (1)	1	1
Gapev (1)	1	0	Gappusin (8)	0.75	0.75
Geinimi (25)	1	0.72	Generic (1)	1	1
GinMaster (23)	0.53	0.69	Glodream (6)	0.83	0.83
Gmuse (1)	1	1	Gonca (2)	1	1
Hamob (3)	1	1	Hispo (1)	1	0
Iconosys (13)	1	1	Imlog (6)	1	1
JSmsHider (1)	1	0	Jifake (4)	1	1
Kidlogger (2)	1	0.5	Kiser (2)	1	1
Kmin (19)	0.95	1	Ksapp (1)	0	0
DroidKungFu (180)	0.96	0.92	Lemon (2)	0	0
LifeMon (1)	1	1	Luckycat (1)	1	1
Mania (1)	1	1	MobileTx (19)	0.95	1
Mobilespy (3)	0.67	0.67	Mobinauten (1)	0.67	0
Moghava (1)	0.67	0	Nandrobox (3)	0.75	1
Nickspy (2)	1	0.5	NickyRCP (1)	1	1
Nisev (1)	0.33	1	Nyleaker (3)	1	0.33
Opfake (188)	0.99	1	PdaSpy (1)	1	1
Penetho (3)	1	1	Placms (3)	1	1
Plankton (44)	0.82	0.95	Proreso (1)	1	1
QPlus (1)	1	1	Raden (2)	1	0.5
RediAssi (1)	1	1	RootSmart (2)	0.67	1
Rooter (1)	1	1	SMSZombie (2)	1	1
SMSreg (12)	0.83	0.83	Sakezon (2)	0.67	1
SeaWeth (2)	1	0.5	SendPay (8)	0.89	1
SerBG (3)	0.5	1	SheriDroid (1)	1	1
SmsWatcher (1)	1	1	Spitmo (3)	1	1
SpyBubble (1)	1	0	SpyHasb (3)	1	1
SpyMob (1)	1	1	SpyPhone (2)	1	0
Spyoo (1)	1	1	Spyset (2)	1	1
Stealer (2)	1	1	Stealthcell (2)	1	0.5
Steek (3)	1	1	Stiniter (2)	1	0.5
Tapsnake (1)	1	1	TigerBot (1)	1	1
Trackplus (2)	1	0.5	Denofow (1)	1	1
Hippo (2)	0	0	Typstu (2)	1	1
Vdloader (3)	0.33	0.33	Vidro (2)	1	1
Xsider (2)	0.67	1	Yzhc (10)	1	0.9
Zitmo (2)	1	1	Zsone (2)	0.5	1

malware variants using control and data flow analysis. In *MILITARY COMMUNICATIONS CONFERENCE, MILCOM 2012*, pages 1–6.

Androguard (2015). `https://code.google.com/p/androguard/`.

Arp, D., Spreitzenbarth, M., Huebner, M., Gascon, H., and Rieck, K. (2014). Drebin: Efficient and explainable detection of android malware in your pocket. In *Proceedings of 21th Annual Network and Distributed System Security Symposium (NDSS)*.

Azab, A., Layton, R., Alazab, M., and Oliver, J. (2014). Mining malware to detect variants. In *5th Cybercrime and Trustwortly Computing Cnference*, pages 44–52.

Battista, P., Mercaldo, F., Nardone, V., Santone, A., and Visaggio, C. A. (2016). Identification of android malware families with model checking. In *International Conference on Information Systems Security and Privacy*. SCITEPRESS.

Bayer, U., Comparetti, P. M., Hlauschek, C., Kruegel, C., and Kirda, E. (2009). Scalable, behavior-based malware clustering. In *NDSS*, volume 9, pages 8–11. Citeseer.

Canfora, G., Mercaldo, F., and Visaggio, C. (2016). Evaluating op-code frequency histograms in malware and third-party mobile applications. *Communications in Computer and Information Science*, 585:201–222.

Canfora, G., Mercaldo, F., and Visaggio, C. A. (2015). Mobile malware detection using op-code frequency histograms. In *Proceedings of International Conference on Security and Cryptography (SECRYPT)*.

Carrera, E. and Erdélyi, G. (2004). Digital genome mapping–advanced binary malware analysis. In *Virus bulletin conference*, volume 11.

Cesare, S. and Xiang, Y. (2011). Malware variant detection using similarity search over sets of control flow graphs. In *IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 181–189.

Chen, X., Andersen, J., Mao, Z. M., Bailey, M., and Nazario, J. (2008). Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 177–186. IEEE.

Dalvik (2015). `http://source.android.com/devices/tech/dalvik/dalvik-bytecode.html`.

Dumitras, T. and Neamtiu, I. (2011). Experimental challenges in cyber security: A story of provenance and lineage for malware. *CSET*, 11:2011–9.

F-Secure (2015). `https://www.f-secure.com/documents/996508/1030743/Threat_Report_H1_2014.pdf`.

Farhadi, M. R., Fung, B., Charland, P., and Debbabi, M. (2014). Binclone: detecting code clones in malware. In *Software Security and Reliability (SERE), 2014 Eighth International Conference on*, pages 78–87. IEEE.

Gascon, H., Yamaguchi, F., Arp, D., and Rieck, K. (2013). Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013*

ACM Workshop on Artificial Intelligence and Security, AISec '13, pages 45–54, New York, NY, USA. ACM.

Giannella, C. and Bloedorn, E. (2015). Spectral malware behavior clustering. In *Intelligence and Security Informatics (ISI), 2015 IEEE International Conference on*, pages 7–12. IEEE.

Goldberg, L. A., Goldberg, P. W., Phillips, C. A., and Sorkin, G. B. (1998). Constructing computer virus phylogenies. *Journal of Algorithms*, 26(1):188–208.

Hu, X., Chiueh, T.-c., and Shin, K. G. (2009). Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 611–620. ACM.

Jang, J., Brumley, D., and Venkataraman, S. (2011). Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 309–320. ACM.

Jilcott, S. (2015). Scalable malware forensics using phylogenetic analysis. In *Technologies for Homeland Security (HST), 2015 IEEE International Symposium on*, pages 1–6. IEEE.

Karim, M. E., Walenstein, A., Lakhotia, A., and Parida, L. (2005). Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, 1(1-2):13–23.

Kinable, J. and Kostakis, O. (2011). Malware classification based on call graph clustering. *Journal in computer virology*, 7(4):233–245.

Kong, D. and Yan, G. (2014). Transductive malware label propagation: Find your lineage from your neighbors. In *INFOCOM, 2014 Proceedings IEEE*, pages 1411–1419. IEEE.

Kruegel, C., Kirda, E., Mutz, D., Robertson, W., and Vigna, G. (2005). Polymorphic worm detection using structural information of executables. In *Recent Advances in Intrusion Detection*, pages 207–226. Springer.

Ma, J., Dunagan, J., Wang, H. J., Savage, S., and Voelker, G. M. (2006). Finding diversity in remote code injection exploits. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 53–64. ACM.

Mercaldo, F., Visaggio, C. A., Canfora, G., and Cimitile, A. (2016). Mobile malware detection in the real world. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016*, pages 744–746.

Nagra, J. and Collberg, C. (2009). *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education.

Rieck, K., Trinius, P., Willems, C., and Holz, T. (2011). Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4):639–668.

Rosenblum, N., Miller, B. P., and Zhu, X. (2011). Recovering the toolchain provenance of binary code. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 100–110. ACM.

Roy, C. K., Cordy, J. R., and Koschke, R. (2009). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495.

Schipka, M. (2007). A road to big money: evolution of automation methods in malware development. *Martin [17]*.

Shang, S., Zheng, N., Xu, J., Xu, M., and Zang, H. (2010). Detecting malware variants via function-call graph similarity. In *2010 5th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 113–120.

Shen, T., Zhongyang, Y., Xin, Z., Mao, B., and Huang, H. (2014). Detect android malware variants using component based topology graph. In *IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 406–413.

Spreitzenbarth, M., Echtler, F., Schreck, T., Freling, F. C., and Hoffmann, J. (2013). Mobilesandbox: Looking deeper into android applications. In *28th International ACM Symposium on Applied Computing (SAC)*.

Walenstein, A. and Lakhotia, A. (2012). A transformation-based model of malware derivation. In *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on*, pages 17–25. IEEE.

Wehner, S. (2007). Analyzing worms and network traffic using compression. *Journal of Computer Security*, 15(3):303–320.

Wu, L., Xu, M., Xu, J. Zheng, N., and Zhang, H. (2013). A novel malware variants detection method based on function-call graph. In *13Th IEEE Joint International Computer Science and Information Technology Conference (JICSIT)*, pages 1–5.

Xiaofang, B., Li, C., Weihua, H., and Qu, W. (2014). Malware variant detection using similarity search over content fringerprint. In *26th Chinese Control and Decision Conference*, pages 5334–5339.

Yu, S., Zhou, S., Liu, L., Yang, R., and Luo, J. (2010). Malware variants identification based on byte frequency. In *Networks Security Wireless Communications and Trusted Computing (NSWCTC), 2010 Second International Conference on*, volume 2, pages 32–35. IEEE.

Zhong, Y., Yamaki, H., and Takakura, H. (2012). A malware classification method based on similarity of function structure. In *Applications and the Internet (SAINT), 2012 IEEE/IPSJ 12th International Symposium on*, pages 256–261. IEEE.

Zhong, Y., Yamaki, H., Yamaguchi, Y., and Takakura, H. (2013). Ariguma code analyzer: Efficient variant detection by identifying common instruction sequences in malware families. In *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*, pages 11–20. IEEE.

Zhou, Y. and Jiang, X. (2012). Dissecting android malware: Characterization and evolution. In *Proceedings of 33rd IEEE Symposium on Security and Privacy (Oakland 2012)*.