ANASTASIA: ANdroid mAlware detection using STAtic analySIs of Applications

Hossein Fereidooni
University of Padua, Italy
Email: hossein@math.unipd.it

Mauro Conti
University of Padua, Italy
Email: conti@math.unipd.it

Danfeng Yao
Department of Computer Science, Virginia Tech, USA
Email: danfeng@cs.vt.edu

Alessandro Sperduti
University of Padua, Italy
Email: sperduti@math.unipd.it

***Abstract*—The number of malware applications targeting the Android operating system has significantly increased in recent years and malicious applications pose a significant threat to Android platform security. We propose ANASTASIA, a system to detect malicious Android applications through statically analyzing applications' behaviors. ANASTASIA provides a more complete coverage of security behaviors when compared to state-of-the-art solutions. We utilize a large number of statically extracted features from various security behavioral characteristics of an application. We built a Machine Learning-based detection framework with high performance detection and acceptable false positive rate. The significance of our work is to develop a lightweight malware detection system for Android-powered smartphones that leverages robust, effective, and efficient features. Besides, in order to assess our solution, we used a reliable, large-scale, and updated malware data-set in terms of diversity and number of malware applications. We evaluated the performance of our proposal on large-scale malware data-set (including 18,677 malware and 11,187 benign apps). Our experimental results show a true positive rate of 97.3% and a false negative rate of 2.7%. These results are better than what are reported by state-of-the-art Android malware detection methods.**

keywords: Android static analysis; Malware detection; Machine Learning.

I. Introduction

Smartphones are continuously replacing more traditional mobile phones. People use those mobile devices for several types of applications, often involving personal information (contacts, emails, agenda, pictures, banking, etc.). According to the Bring Your Own Device (BYOD) policy, adopted by many companies, the very same personal device is also used to access the IT infrastructure of the company where the smartphone owner is employed. In this scenario, the security of these devices as well as the assets that they allow access to are at stake. In fact, the security for smartphones still needs a thorough understanding (as shown from several attacks, e.g., the one in [10]). The effective way of enforcing security on those devices is still subject of investigation and there are further room for improvement. To address this issue, we can leverage various techniques to analyze and detect Android malware applications. Static analysis refers to extract and analyze information about an application from the binary, source code or other associated files. Static analysis can be performed before running the application for the first time. However, the mentioned method is limited because of the obfuscation techniques and might not be able to deal with the malware which changes its code without changing functionality (e.g., polymorphic malware).

Static analysis is beneficial on memory-limited Android-powered devices because the malware is not executed and only analyzed. For these reasons, we will concentrate on the lightweight approach, we advocate the static analysis.

We present a detection approach to hunt malicious Android applications that achieves a higher detection performance than previously reported classification methods. In particular, our main contributions are as follows:

- We presented an Android malware detection method that uses several informative features with good discriminative power to discern benign from malware apps. To extract these features, we designed and built a tool named *uniPDroid*, written in Python programming language.
- We performed an extensive static analysis on large-scale well-labeled data-set of 29,864 Android applications.
- We used several Machine Learning classification algorithms including ensemble, eXtreme Gradient Boosting and Deep Learning to discover the most performant one in terms of accuracy and speed. Our experimental evaluations show that our proposed detection method is very effective and efficient. It obtained a true positive rate in detecting malware applications as high as **97.3%** and false negative rate as low as **2.7%**.

The rest of the paper is organized as follows. Section II illustrates the design and implementation of our proposal: ANASTASIA. In this section, we explain system architecture. Also, we give a description of data-set composition and feature extraction procedure. Then, we describe all the details to build our classifiers. We evaluate our proposed method and compare its performance against several malware detection schemes in the literature in Section III. Finally, we conclude the paper in Section IV.

II. Design and Implementation

In this section, we describe design and implementation of the ANASTASIA. The goal of our system is to effectively and efficiently detect malicious Android applications. To this end, we extended Androguard tool and built the uniPDroid, a static analysis tool written in Python programming language. Our

978-1-5090-2914-3/16/$31.00 ©2016 IEEE

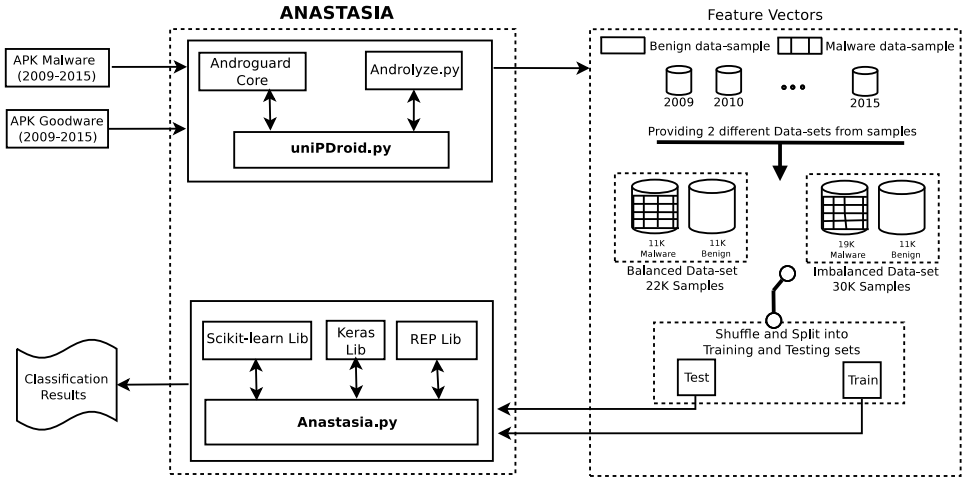

Fig. 1: ANASTASIA System Overview.

proposed method uses this tool to extract several informative features representing characterization of the application and leverages several Python Machine Learning libraries to build the most performant classifier to perform classification task. The main components of the ANASTASIA are illustrated in Figure 1. In particular, the system consists of two modules: (i) Feature Extraction Module, and (ii) Machine Learning Classification Module. The feature extraction module includes three components. The `uniPDroid.py` is the main component within this module extracting informative features from an app while `Androguard` and `Anadrolyze.py` are auxiliary components providing support for performing feature extraction task. The Machine Learning classification module leverages several Machine Learning packages to perform classification task. The main component within this module is the `anastasia.py`. The `Scikit-learn`, `Keras` and `REP` packages provide different classification algorithms and some helper functions for performance evaluation.

A. Data-set Composition

In this section, we detail how our data-set has been composed. In order to conduct an extensive analysis on malware detection, we collected a set of large well-labeled malware applications. The data-set that we used during our evaluation is composed of 29,864 Android apps collected from several work in the literature [7], [15], [25], [28], hence consisting of apps released over period of seven years from year 2009 to 2015. Table I shows the details of the data-set composition.

Repository	Malware	Benign	Total
Genome	1,260	–	1,260
Drebin	5,560	–	5,560
M0Droid	193	200	393
VirusTotal	11,664	10,987	22,651
Resulting data-set	**18,677**	**11,187**	**29,864**

TABLE I: Data-set composition

B. Feature Extraction

In this section, we describe the feature engineering conducted in this work. We aimed at extracting as many features as possible in order to allow the Machine Learning algorithms to select the most informative features to perform the classification task. Using uniPDroid tool, we mainly extracted the features from bytecode and converted these features into binary feature vectors including 560 features. We collected a feature set from disassembled code as follow:

- *Intents:* inter-process and intra-process communication on Android is mainly performed through intents. The intent is an abstract description of an operation to be performed and allowing information about events to be shared among different components and applications. We extract all intents used in Android app as a feature set because malwares often listen to specific intents. For instance, `BOOT_COMPLETED` is a typical example of an intent message involved in malicious apps, which is used to trigger malicious activity directly after rebooting the smartphone.
- *Used permissions:* a significant part of Android's built-in security is the permissions system. Permissions allow an application to access potentially dangerous API calls. Since the used permissions in `.dex` file provides a more general view on the behavior of an application rather than permissions declared in manifest file. We extract and include them to the feature set (e.g., INTERNET).
- *System Commands:* system commands are used in the malware to run root exploit code or download and install additional executable files. Since system command can provide us with valuable information to detect malicious behavior, we extract and include them in the feature set. The authors in [24] listed the most commonly used system commands in malicious apps (e.g., `chmod, su, sh, ps`). These commands are executed after the malware achieves root privilege on the device.
- *Suspicious API calls:* some of the API calls are able to access to sensitive resources or information of the smartphone. These type of API calls are frequently seen in malware samples and can result in malicious behavior. In order to obtain a deeper understanding of the function-

ality of an application, we collected these API calls and gathered in the feature set (e.g., `sendTextMessage`, `Runtime.exec`). The authors in [24] mentioned the most commonly used API calls in malicious apps.

- *Malicious Activities:* we investigated whether an Android app is capable of performing such malicious activities through Dalvik bytecode analysis. We considered different kinds of information that malicious apps are able to harvest from smartphones. We include these features in our feature set in order to discern benign applications from malicious ones. Due to space limit, we have listed some of these features including: reading the IMEI, loading dynamic, and reflection code, opening a socket, making a call and performing encryption and message digest.

C. Feature Selection

A large number of features, some of which redundant or irrelevant might present several problems such as misleading the learning algorithm, and increasing model complexity. To mitigate above-mentioned problems feature selection techniques are used. We used a technique leveraging ensemble of randomized decision trees (i.e., Extra Trees-Classifier) [21] for determining the feature importances. The features are considered unimportant are discarded, if the corresponding feature importance values are below the provided threshold parameter (e.g., Mean).

D. Classification Models

Our objective is to build a promising model to classify unknown Android apps as either benign or malware. To this end, we have employed several algorithms for the classification such as XGboost [9], Adaboost [6], RandomForest [5], SVM with RBF kernel [3], K-NN [4], Logistic Regression, Naive Bayes, Decision Tree classifiers [21], and Deep Learning [22]. Since Deep Learning is a growing trend in Machine Learning, we briefly introduce this technique. Deep Learning refers to artificial neural networks that are composed of many layers. There are two key parameters while building the Deep Learning model: (i) the number of layers and, (ii) the number of neurons in each layer. The first layer is a type of visible layer called an input layer. This layer contains an input node for each of the entries in our feature vector. Input layer's nodes connect to a series of hidden layers. In the most simple terms, each hidden layer is an unsupervised Restricted Boltzmann Machine (RBM) where the output of each RBM in the hidden layer sequence is used as input to the next layer. Finally, we have our another visible layer called the output layer. This layer contains the output probabilities for each class label. The layers are dense layer which means regular fully connected layer. During the training procedure, we used Stochastic Gradient Descent (SGD) as an optimizer and `binary_crossentropy` as a optimization score function. We aimed at finding the most performant classifier in terms of accuracy and speed. To perform a learning task allowing classification of apps into the malware and benign classes, we have to tune hyper-parameters of classification algorithms. Many Machine Learning classification algorithms have hyper parameters and these parameters modify the nature of the model. So, proper settings of these parameters can be critical to optimize the performance of the model for a specific data source. We tuned the hyper-parameters of above-mentioned classifiers through *Grid-search* and *Cross-validation* processes implemented in Scikit-learn Machine Learning package [21]. Table II shows the best parameters. Having the best hyper-parameter selected, we evaluated the performance of all classification algorithms through 10-fold Cross-validation to find out the most performant classifier. In the following, we explain the details of selecting the most performant classifier to incorporate in our detection framework.

We prepared a balanced data-set including 11,000 malware and 11,000 benign samples. We shuffled and split the data-points into training and testing sets of 20,000 and 2,000 samples, respectively. We conducted 10-fold cross validation on training set. In 10-fold cross-validation, the data is randomly partitioned into 10 equal size subsamples. Of the 10 subsamples, a single subsample is retained as the validation data for testing the model, and the remaining 9 subsamples are used as training data. The process is then repeated 10 times, with each of the 10 subsamples used exactly once as the validation data. The 10 results from the folds can then be averaged to produce a single estimation. We also repeated this experiment for an imbalanced data-set consisting of 18,766 malware and 11,187 benign samples. In the latest experiment, we split the data into training and testing sets, each one 26,864 and 3,000 samples, respectively. Table III shows the results obtained for each classification algorithm for balanced and imbalanced data-sets.

Classifier	**The best parameters**
SVM	kernel="rbf", C=1, gamma=0.001
DT	max_depth=6
LR	C=1
NB	- -
RF	n_estimators=600, max_depth=8
KNN	n_neighbors=5
Adaboost	n_estimators= 600, learning_rate=1
XGboost	n_estimators=600, eta=0.1, max_depth=12
Deep Learning	nEpoch=600, lr=0.1, decay=1e-6, momentum=0.9, num_hidden_layers=6, layer_size=[3200,1600,800,400,200,100]

TABLE II: Parameter tuning via Grid-search and Cross-validation

Our analysis shows that XGBoost model lead to a better accuracy compared to the other models. We have exploited this classifier in our malware detection framework.

III. Evaluation and Benchmark

In this section, we report on the experimental evaluation of our malware detection system. Our aim is to answer these research questions:

- **Q1:** How well are accuracy scores of our methodology in terms of F1-score, Recall and Precision in both balance and imbalance class of problems?

Classifier	F1-Score Imbalanced	F1-Score Balanced
SVM	0.94 (+/- 0.01)	0.93 (+/- 0.01)
DT	0.83 (+/- 0.02)	0.82 (+/- 0.02)
LR	0.91 (+/- 0.01)	0.91 (+/- 0.01)
NB	0.87 (+/- 0.01)	0.85 (+/- 0.02)
RF	0.96 (+/- 0.01)	0.95 (+/- 0.01)
KNN	0.94 (+/- 0.01)	0.93 (+/- 0.01)
Adaboost	0.95 (+/- 0.01)	0.95 (+/- 0.01)
Deep Learning	0.96 (+/- 0.003)	0.95 (+/- 0.002)
XGboost	0.97 (+/- 0.01)	0.96 (+/- 0.01)

TABLE III: 10-fold Cross-validation scores using imbalanced and Balanced data-sets.

- **Q2:** How much are the true positive rate, the false positive rate, and false negative rate achieved by our proposed method?

A. Experimental Setup on balanced data-set

Apart from XGBoost classifier, the most performant classifier according to 10-fold CV results, we trained others classifiers on training set (20,000 samples) and then tested their performance against testing set (2,000 unseen samples). We did this to double-check that XGBoost classification algorithm outperforms other classifiers. We computed accuracy measures, time taken to build the model (T_{Train}), time taken to evaluate the model over testing set (T_{Test}), and Confusion Matrix of the classifier on balanced data-set. Due to space limit, we just have shown the results achieved from this experiment for XGboost classifier as shown in Tables IV and V.

	Pre.	Rec.	F1	Supp.	T_{Train}	T_{Test}
Benign	95	95	96	994		
Malware	96	95	96	1006		
Avg / Total	96	95	**96**	2000	60.6	0.29

		Predicted Label: Benign	Predicted Label: Malware
Actual	Benign	TN: **956**	FP: **38**
	Malware	FN: **51**	TP: **955**

TABLE IV: Classification Report and Confusion Matrix of XGboost on balanced data-set.

	Pre.	Rec.	F1	Supp.	T_{Train}	T_{Test}
Benign	96	97	96	1100		
Malware	97	97	97	1900		
Avg / Total	97	97	**97**	3000	72	0.4

		Predicted Label: Benign	Predicted Label: Malware
Actual	Benign	TN: **1035**	FP: **65**
	Malware	FN: **51**	TP: **1849**

TABLE V: Classification Report and Confusion Matrix of XGboost on imbalanced data-set.

B. Experimental Setup on imbalanced data-set

In this experiment, we trained the classifiers on training set (26,677 samples) and then tested their performance against testing set (3,000 unseen samples). Due to space limit, we just mentioned the results obtained by the most performant classifier. In the following, we put our emphasis on some of malware detection schemes outperforming others while using static analysis approach. In Table VI, we compare our proposed method with the most performant approaches in the literature. The comparison is based on evaluation metrics such as precision, recall, F-1 score, accuracy scores, TPR, FNR, FPR as well as data-set size used in the experiments. We have to point out that in order to show the improved accuracy of our technique compared to other approaches, it would be more convincing to compare the different approaches on exactly the same data-set. Most of the methods we mentioned in our comparison table have used the same APK files or at least have some APK samples in common. The difference steams from extracting different features from those APK files. The resulting feature sets provided by these methods differ in terms of type of features and number of features. For instance, DroidMat uses permissions, intents, app components, and API calls as features to perform classification task at hand. Drebin exploits permissions, H/W and app components, filtered intents, API calls, and URLs. DroidAPIMiner leverages permissions and API calls. Puma just uses permissions as features. AndroSAT have extracted permissions, app components, and some API calls from Java source code and smali files. ADAGIO exploits function call graph extracted from apps. AndroTracker uses intents, permissions, API calls, system commands. It is worth mentioning that our proposed method outperforms other state-of-the-art approaches. However, some of the schemes that we investigated have not reported these performance metrics in their experiments. As for testing accuracy score, DroidMat test score is just 1% higher than our method, while their testing set size is not comparable with ours, specifically in terms of number of malware samples. The test accuracy score of DroidAPIMiner is 3% better than our method. However, for evaluating the performance on such an imbalanced data-set, the authors should have mentioned precision, recall and F1-score. In addition to this, DroidAPIMiner was built on a KNN classification algorithm that introduces runtime overhead much higher than our proposed method. It takes on average about 10 sec for K-NN classification algorithm used in DroidAPIMiner to classify an APK file as either benign or malicious [1].

IV. Conclusions and Future Work

In this paper, we proposed ANASTASIA, a Machine Learning-based malware detection using static analysis of Android applications. To this end, we designed a tool, uniPDroid, to extract as many informative features as possible from Android applications. We trained several classification algorithms to find out the most performant ones in terms of accuracy and speed. We performed an extensive Grid-search analysis along with Cross-validation to tune the hyper-parameter of classifiers to obtain as much detection perfor-

Detection scheme	Malware	Benign	Total	CV	Test	Pre.	Rec.	F-1	TPR	FNR	FPR
Puma, 2012 [20]	249	1,811	2,060	86.4%	N/A	N/A	N/A	N/A	91%	N/A	19%
Androsat, 2014 [17]	N/A	N/A	1,932	95%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AndroTracker, 2015 [12]	4,554	51,179	55,733	98%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Allix et. al, 2014 [2]	1,200	51,800	52,000	91%	N/A	94%	91%	91%	N/A	N/A	N/A
DroiMat, 2012 [26]	238	1,500	1,738	N/A	97%	96%	87%	92%	87%	13%	0.4%
Sato et. al, 2013 [13]	130	235	365	N/A	90%	N/A	N/A	N/A	N/A	N/A	N/A
Drebin, 2014 [7]	5,560	123,453	129,016	N/A	93.8%	N/A	N/A	N/A	N/A	N/A	1%
DroidAPIMiner, 2013 [1]	3,978	16,000	19,978	N/A	99%	N/A	N/A	N/A	97%	3%	2.2%
Droid-Sec, 2014 [23]	300	200	N/A	N/A	96.5%	N/A	N/A	N/A	N/A	N/A	N/A
ADAGIO, 2013 [11]	12,158	135,792	147,950	N/A	89%	N/A	N/A	N/A	N/A	N/A	1%
ANASTASIA*	**11,000**	**11,000**	**22,000**	**96%**	**96%**	**96%**	**96%**	**96%**	**95%**	**5%**	**3.8%**
ANASTASIA+	**18,677**	**11,187**	**29,864**	**97%**	**97%**	**97%**	**97%**	**97%**	**97.3%**	**2.7%**	**5%**

* Balanced data-set
+ Imbalanced data-set

TABLE VI: Performance comparison of our method against state-of-the-art.

mance as possible. We selected the most performant classifier to incorporate in our framework. We obtained accuracy score of 97% in detection of unseen malware for imbalanced data-sets proving the robustness of features extracted . In addition to this, we achieved true positive rate as high as 97.3% and false negative rate as low as 2.7%. As a future work, we will extend our Android application analysis and combine static analysis with dynamic analysis to overcome to limitation associated with static analysis that we already explained. We will extract more features related to behavior of Android applications such as CPU and Memory consumption, Network traffic activities, Inter-Process Communications and system calls made by applications so as to interact with Android OS.

ACKNOWLEDGMENT

Hossein Fereidooni is supported by Deutsche Akademische Austauschdienst (DAAD fellowship). Mauro Conti is also supported by a Marie Curie Fellowship funded by the European Commission (agreement PCIG11-GA-2012-321980). This work is also partially supported by the EU TagItSmart! Project (agreement H2020-ICT30-2015-688061).

REFERENCES

[1] Y. Aafer, W. Du, and H. Yin. Droidapiminer: Mining API-level features for robust malware detection in android. *9th International ICST Conference, SecureComm 2013*, pages 86–103, Sydney, 2013.

[2] K. Allix, T. F. Bissyandé, Q. Jérome, J. Klein, R. State, and Y. L. Traon. Empirical assessment of machine learning-based malware detectors for android. *Journal of Empirical Software Engineering*, 2014.

[3] V. Vapnik. The Nature of Statistical Learning Theory. Springer-Verlag, NY, 1995.

[4] D. W. Aha, D. Kibler, and M. K. Albert. Instance-Based Learning Algorithms. Machine Learning, 6:3766, 1991.

[5] Leo Breiman. Random Forests. , University of California, Berkeley, Journal of Machine Learning, Vol. 45 Issue 1, pp 5 - 32, October, 2001.

[6] Yoav Freund, and E. Schapire. Experiments with a New Boosting Algorithm. In*Proc. of the Thirteenth International Conference*, 1996.

[7] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck. Drebin: Efficient and explainable detection of android malware in your pocket. In *Proc. of 17th Network and Distributed System Security Symposium*, Feb. 2014.

[8] M. Grace, Z. Yajin, Z. Qiang, Z. Shihong, and J. Xuxian. Riskranker: scalable and accurate zero-day android malware detection. In *Proc. of the 10th International Conference on Mobile Systems, Applications, and Services*, 2012.

[9] T. Chen. https://github.com/dmlc/xgboost.

[10] E. Fernandes, B. Crispo, and M. Conti. FM 99.9, radio virus: Exploiting FM radio broadcasts for malware deployment. In *(IEEE) Transactions on Information Forensics and Security*, 2013.

[11] F. Gascon, F. Yamaguchi, and D. Arp. Structural detection of android malware using embedded call graphs. In *Proc. of the ACM workshop on Artificial intelligence and security*, pp. 45–54, ACM NY, USA, 2013.

[12] H. Kang, J. wook Jang, A. Mohaisen, and H. K. Kim. Detecting and classifying android malware using static analysis along with creator information. *International Journal of Distributed Sensor Networks*, 2015.

[13] R. Sato, D. Chiba, S. Goto. Detecting Android Malware by Analyzing Manifest Files. In *Proc. of the Asia-Pacific Advanced Network*, pp. 23-31, 2013.

[14] D. Koundel, S. Ithape, V. Khobaragade, and R. Jain. Malware classification using naïve bayes classifier for android os. *The International Journal of Engineering And Science*, 3:59–63, 2014.

[15] M0droid. http://m0droid.netai.net/modroid/.

[16] mlxtend. https://github.com/rasbt/mlxtend.

[17] S. Oberoi, W. Song, and A. M. Youssef. Androsat: Security analysis tool for android applications. *The 8th International Conference on Emerging Security Information, Systems and Technologies*, 2014.

[18] N. Peiravian and X. Zhu. Machine learning for android malware detection using permission and api calls. In *Proc. of the IEEE 25th International Conference on Tools with Artificial Intelligence*, pp. 300–305, 2013.

[19] G. Russello, M. Conti, and B. C. and E. Fernandes. Moses: Supporting operation modes on smartphones. *In Proc. of the 17th ACM Symposium on Access Control Models and Technologies*, pp. 3–12, Newark, US, June 20-22, 2012.

[20] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. Bringas, and G. Alvarez. Puma: permission usage to detect malware in android. In *Proc. of the CISIS'12-ICEUTE'12-SOCO'12*, 2012.

[21] Scikit-learn. https://github.com/scikit-learn/.

[22] Keras. http://keras.io/#keras-deep-learning-library-for-theano-and-tensorflow.

[23] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, Yibo Xue. Droid-Sec: deep learning in android malware detection. *CCR August 2014.*

[24] Seo, Gupta, M. Sallam, Bertino, and Y. K. Detecting mobile malware threats to homeland security through static analysis. *Journal of Network and Computer Applications*, pp. 43–53, 2014.

[25] VirusTotal. http://www.virustotal.com.

[26] D. Wu, C. Mao, T. Wei, H. Lee, and K. Wu. Droidmat: Android malware detection through manifest and api calls tracing. *In Proc. of the 2012 Seventh Asia Joint Conference on Information Security*, pp. 62–69, Washington, DC, USA, 2012.

[27] M. Zheng, M. Sun, and S. Lui. Droidanalytics: A signature based analytic system to collect, extract, analyze and associate android malware. 2013.

[28] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. *2012 IEEE Symposium on Security and Privacy*, pp. 95–109, 20-23 May, 2012.