MADLIRA - a tool for Android malware detection

Khanh Huu The Dam[1] and Tayssir Touili[2]
[1]*Nha Trang University, Vietnam*
[2]*LIPN, CNRS and University Paris 13, France*

Keywords: malware detection, Android, static analysis

Abstract: Today, there are more threats to Android users since malware writers are changing their target to explore the weakness of Android devices, in order to generate malicious behaviors. Thus, detecting Android malwares is becoming crucial. We present in this paper a tool, called MADLIRA (MAlware Detection using Learning and Information Retrieval for Android). This tool implements two static approaches: (1) apply Information Retrieval techniques to *automatically extract* malicious behaviors from a set of malicious and benign applications, (2) apply learning techniques to *automatically learn* malicious applications. Then, in both cases, MADLIRA can classify a new Android application as malicious or benign.

1 INTRODUCTION

The number of new malwares increased by 36 percent in one year from 2014 to 2015. It is estimated that there are more than one million of new pieces of malwares released everyday (Sym, 2016). According to the report of Kaspersky Lab in the third quarter of 2017, there are more than 1.5 million malicious packages detected on mobiles (Kas, 2018). The last years, the number of attacks on mobiles has increased enourmeously, using various types such as backdoors, cryptomining, fake apps, banking trojans, etc. (McA, 2019). Consequently, there are more threats to Android users. Thus, the challenge is to detect malicious behaviors in Android applications. However, most of industry anti-viruses are based on the signature matching technique where a scanner will search for specific elements (called signatures) like permission requirements, package names and class names or the segmentations of bytecode and certain strings in the Android applications. If an application contains such a signature, it is marked as malicious. If not, it is marked as benign. This signature matching technique is not very robust. Indeed, malware writers may implement some obfuscation techniques, e.g., renaming, inserting functions (Rastogi et al., 2013; Zheng et al., 2013; Maiorca et al., 2015; Preda and Maggi, 2016) to change the structure of a malware while keeping its same behaviors so that the known signatures cannot be used to recognize it.

To avoid this issue, several works (Burguera et al., 2011; Dimjaševic et al., 2015; Canfora et al., 2015; Jang et al., 2016; Malik and Khatter, 2016) use dynamic analysis to analyze the behaviors of the Android applications instead of its syntactic signatures. In this approach, the behaviors are dynamically observed while running the application on an emulated environment. However, by the dynamic analysis technique, it is hard to trigger the malicious behaviors in a short period since they may require a delay or only show up after some interaction of users.

To overcome these limitations, one needs to analyse the *behaviors* of a program *without* executing it. To this aim, we consider in this paper API calls in Android applications to model the malicious behaviors in a static way. Indeed, API functions are mediators between programs and their running environment. They are used to access or modify the system by malware authors. Therefore, API functions are crucial to specify malicious behaviors. Thus, in this work, we model a program using an API call graph, which is a directed graph whose nodes are API functions, and whose edges specify the execution order of the API function calls: an edge (f, f') expresses that there is a call to the API function f followed by a call to the API function f'. For example, let us look at a typical behavior of an Android trojan SMS spy. The smali code of this behavior is given in Figure 1. This behavior consists in collecting the phone id and then sending this data via a text message. This task is done by a sequence of API calls. First, the function getDeviceId() is called at line 5 to collect the phone id. Then, the TelephonyManager object is gotten by calling getDefault() at line 19. Finally, the phone id is sent to an anonymous phone number via a text message by calling sendTextMessage() at line 25.

To represent this behavior, we use a malicious API graph. Figure 2 shows the malicious API graph of this behavior. The edges express that a call to the function Landroid/telephony/TelephonyManager; -> getDeviceId() is followed by the calls to the functions Landroid/telephony/gsm/SmsManager; -> getDefault() and Landroid/telephony/gsm/SmsManager; -> sendTextMessage().

```
1  .class public Lcom/km/MainActivity;
2  .super Landroid/app/Activity;
3  .method public onCreate(Landroid/os/Bundle;)V
       ...
4      invoke-direct {v0, v1}, Ljava/lang/StringBuilder;-><init>()V
5      invoke-virtual {v2}, Landroid/telephony/
                           TelephonyManager;->getDeviceId()
6      move-result-object v1
7      iput-object v1, p0, Lcom/km/MainActivity;->data
8      return-void
9  .end method

10 .method public OnStart()V
       ...
11     iget-object v1, p1, Lcom/km/MainActivity;->number
12     iget-object v2, p2, Lcom/km/MainActivity;->data
13     invoke-virtual {p0,v1,v2}, Lcom/km/SendMessage;->Send()
14     return-void
15 .end method
16 .class public Lcom/km/SendMessage;
17 .method public Send(Ljava/lang/String;Ljava/lang/String;)V
18     const/4 v2, 0x0
19     invoke-static {}, Landroid/telephony/gsm/
                                 SmsManager;->getDefault()
20     move-result-object v0
21     iget-object v4, p0, Lcom/km/SendMessage;->mSendPI
22     move-object v1, p1
23     move-object v3, p2
24     move-object v5, v2
25     invoke-virtual/range {v0 .. v5}, Landroid/telephony/gsm/
                                 SmsManager;->sendTextMessage()
26     return-void
27 .end method
```

Figure 1: A piece of smali code of an Android trojan SMS spy.

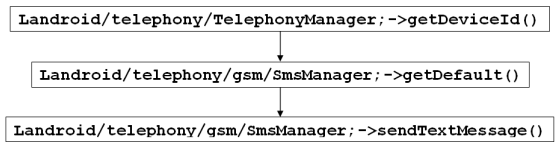

Figure 2: A malicious API graph of an Android trojan SMS spy.

Using this API call graph representation, we implement two static approaches in a tool, called MADLIRA, for Android malware detection: (1) apply the Information Retrieval techniques of (Dam and Touili, 2017a) to automatically extract malicious behaviors from a set of malicious and benign Android applications, and use the extracted malicious behaviors to detect malwares; and (2) apply the learning techniques of (Dam and Touili, 2017b) to automatically learn malicious Android applications. We applied our tool on 3518 Android malwares and 1118 Android benwares, we obtained a detection rate of 98.76% with 0.24% false alarms. We present our tool MADLIRA in this paper. Our tool can be found in https://lipn.univ-paris13.fr/~dam/tool/androidTool/MADLIRA.html

2 RELATED WORK

Similar to our presentation, (Burguera et al., 2011; Gascon et al., 2013; Jang et al., 2016; Song and Touili, 2014; Dam and Touili, 2019b) use API calls to represent the malicious behaviors of malware. (Gascon et al., 2013) represent the applications by function call graphs where nodes correspond to function calls and edges connect the callers to the callees. This model is different from our API call graph where nodes correspond to API function calls and edges specify the execution order of API functions in the application, i.e., it allows the connection of two functions which have the same callers while (Gascon et al., 2013) do not allow that connection. Moreover, our tool allows the extraction of Android malicious behaviors.

In (Burguera et al., 2011; Jang et al., 2016), dynamic analysis is applied to capture the behaviors of an application via system calls. Crowdroid in (Burguera et al., 2011) generates the feature vector for each Android application by counting the number of system calls which are required during its execution. Andro-dumpsys in (Jang et al., 2016) monitors the API calls and its parameters and store them in a profile. As we have mentioned before, dynamic analysis is limited by a time interval. Our analysis is done in a completely static way.

In (Song and Touili, 2014), the authors apply static analysis to model Android applications and then apply model checking for malware detection. Their approach requires the manual specification of malicious behaviors (logic formulas), after a tedious manual study of the code. In contrast, our tool can *automatically* extract malicious behaviors of Android malwares and then use these malicious behaviors to detect malware.

Similar to the STAMAD tool (Dam and Touili, 2019a), our tool MADLIRA allows to learn and extract malicious behaviors. However, STAMAD is implemented for PC malwares whereas our tool MADLIRA tackles Android malwares.

3 BACKGROUND

Given a set of malicious and a set of benign Android applications, MADLIRA first extracts from every program its corresponding API call graph. To perform this step, we apply the techniques of (Dam and Touili, 2017a; Dam and Touili, 2017b) that compute this graph by performing a kind of reachability analysis on the Control Flow Graphs of the programs. Then, MADLIRA can (1) either automatically extract

malicious behaviors using Information Retrieval techniques, or (2) apply machine learning techniques to automatically learn and detect malwares. Then, in both cases, MADLIRA can classify a new given unseen application as malicious or benign. In this section, we will present the main ideas behind these two approaches that are implemented in MADLIRA.

3.1 Extraction of Malicious Behaviors

Given a set of malicious and a set of benign Android applications, MADLIRA implements the idea of (Dam and Touili, 2017a), to automatically extract the malicious behaviors of the malwares. It first extracts the API call graph corresponding to each application, then, MADLIRA's goal is to compute a malicious API call graph that represents the behaviors that are present in the malicious programs but not in the benign ones. To this aim, it will extract the subgraphs which are relevant to the malicious graphs but not relevant to the benign ones. A relevant subgraph contains nodes and edges that are crucial to the malicious API call graphs. This problem can be seen as an Information Retrieval (IR) problem, where the goal is to retrieve relevant items and reject nonrelevant ones. One of the most efficient techniques in information retrieval is the TFIDF[1] scheme. It was widely applied for document extraction by the IR community in web searching, text searching, image searching, etc. (Dam and Touili, 2017a) applied these IR TFIDF techniques to extract malicious API call graphs from a set of Android malwares and benwares. MADLIRA applies the techniques of (Dam and Touili, 2017a) to extract malicious API call graphs: it associates to each node and each edge of the API call graphs in the malicious and benign applications a weight using the formulas of (Dam and Touili, 2017a). These weights are computed from the occurrences of terms (nodes/edges) in benwares and malwares. These weights allow to measure the relevance of each term (nodes/edges) to malwares and benwares. The higher the weight is, the more relevant the term is to malwares. Then, the malicious API call graph is constructed from the edges and nodes that have the highest weight. Finally, MADLIRA uses the automatically extracted malicious behavior specification to determine whether a new unknown program is malicious or not by performing a kind of product between the extracted malicious API call graph, and the new program's API call graph. More details about this approach can be found in (Dam and Touili, 2017a).

[1]TFIDF stands for Term Frequency and Inverse Document Frequency

3.2 Learning Malicious behaviors

In our second approach, we implement the kernel based support vector machine technique on API call graphs to compute a function h which classifies Android malwares from Android benign applications. The choice of support vector machines is motivated by the fact that they are very suitable for nonvectorial data (graphs in our setting), whereas the other well-known learning techniques like artificial neural network, k-nearest neighbor, decision trees, etc. can only be applied to vectorial data. This method is highly dependent on the choice of kernels. A kernel is a function which returns similarity between data. In MADLIRA, we use a variant of the random walk graph kernel that measures graph similarity as the number of common paths of increasing lengths. To implement this graph kernel support vector machine approach to learn Android malwares, MADLIRA applies the ideas and details of (Dam and Touili, 2017b).

4 MADLIRA DESCRIPTION

MADLIRA takes as input a set of Android malwares and a set of Android benwares and can either (1) extract a malicious API graph representing the malicious behaviors of the Android malwares in the set; or (2) learn to classify Android malwares without extracting the malicious behaviors. These phases are called the training phases. Then, given a new Android application, MADLIRA checks whether it is malicious or not.

MADLIRA has two main components: TFIDF component, which extracts the malicious behaviors and uses these malicious behaviors to check whether a new application is malicious or not (read Section 3.1 for more details), and SVM[2] component, which applies random walk graph kernel based support vector machines to classify malwares from benign applications (read Section 3.2 for more details). MADLIRA consists of three modules:

Module 1: Android Application Modeling

This module takes as input an Android application (an APK file). It first applies the apktool (Apk, 2016) to decompile this APK file to smali codes. Then, these smali codes are transformed to the control flow graphs of the Android application. Finally, the Graph Computation component takes as input the Android API

[2]SVM stands for Support Vector Machine

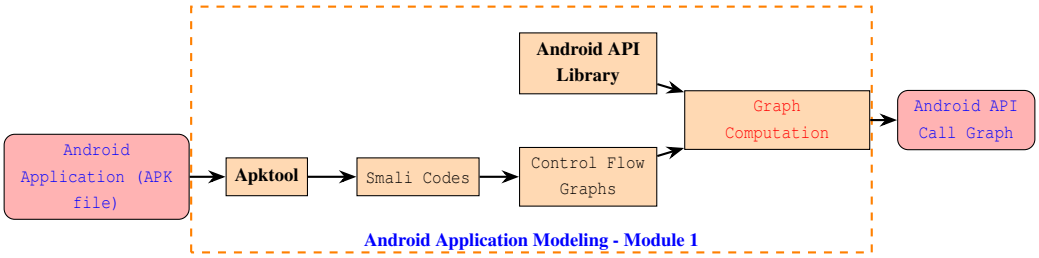

Figure 3: Android Application Modeling - Module 1.

Library and the control flow graph of the Android application to compute the Android API call graph by the algorithm of (Dam and Touili, 2017a).

Module 2 : Extraction of Malicious Behaviors.

This module consists in two phases: the extraction phase, i. e., the extraction of malicious behaviors, and the detection phase, i. e., the malicious behavior detection. In the extraction phase, it takes as input a set of malwares and a set of benwares. After applying the first module to extract their corresponding API call graphs, these graphs are fed to the Malicious Graph Computation component to compute the malicious API graph. This component implements the TFIDF weighting term scheme introduced in Section 3.1 to compute the malicious behaviors. It outputs malicious API graphs representing the malicious behaviors. This phase will be called "training phase".
In the detection phase, this module takes as input an Android application (an APK file) and applies the first module to compute its corresponding API call graph. Then, it checks whether the program's graph contains any malicious behavior from the malicious API graphs (the output of the "training phase") or not. If this program contains any malicious behavior, the output is "Malicious!". Otherwise, the output is "Benign!".

Module 3: Learning Malicious Behaviors.

This module implements the learning technique described in Section 3.2. It consists in two phases: the learning phase and the detection phase. In the learning phase, it takes as input a set of malwares and a set of benwares. It first applies the first Module to compute their corresponding API call graphs. Then, these API call graphs are fed to the SVM training component, i.e., LIBSVM (Chang and Lin, 2011), to compute a SVM training model.
In the detection phase, it takes as input an Android application (an APK file), applies the Application Modeling Module to compute its corresponding API call graph. Then, it uses SVM classifier with the training model (the output of the first phase) to classify the program either" Malicious!" or "Benign!".

5 EXPERIMENTS

To evaluate our tool, we use a dataset of 1118 Android benwares, which are collected from the website apkpure.com, and 3518 Android malwares which are gotten from the Drebin dataset (Arp et al., 2014). MADLIRA gives promising results:

Extraction of malicious behaviors. To evaluate the performance of Module 2, we first applied our tool to automatically extract a malicious API graph from a set of 1900 Android malwares and 704 Android benwares. The obtained malicious API graph is then used for malware detection on a test set of 1618 Android malwares and 414 Android benwares. We obtained encouraging results: a detection rate of 96.6% with 15.7% false alarms.

Learning malware. To evaluate the performance of Module 3, we randomly split the dataset into two partitions, a training and a testing partition. For the training partition, the quantity of malwares and benwares is balanced with 704 samples for each, this will allow us to compute the SVM classifier. The test set consisting of 2814 Android malwares and 414 Android benwares, is used to evaluate the classifier. Using the training set, we compute the training model. Then, we apply this training model to classify Android malwares on the test set and obtain a detection rate of 98.76%, with 0.24% of false alarms.

6 EXAMPLES OF MALICIOUS BEHAVIORS

In this section, we present some malicious behaviors that were *automatically* extracted by MADLIRA.

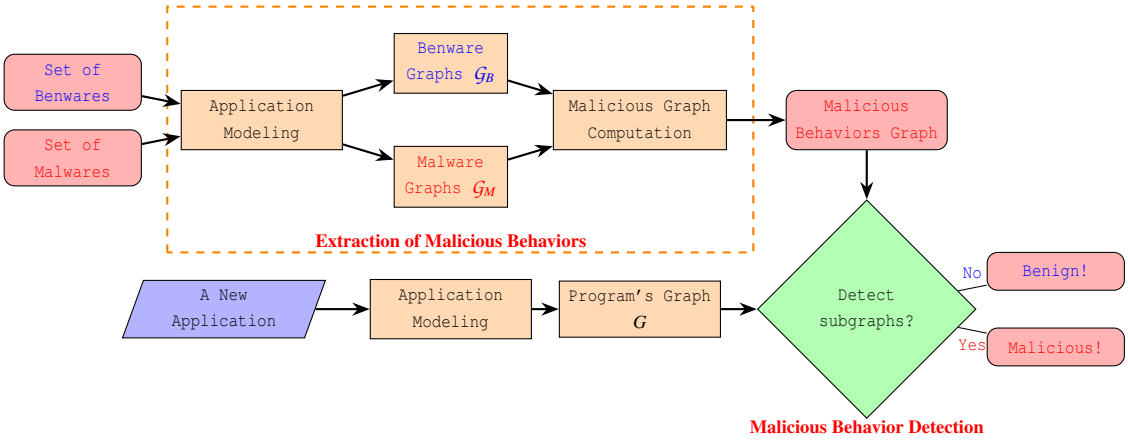

Figure 4: Extraction of Malicious Behaviors - Module 2.

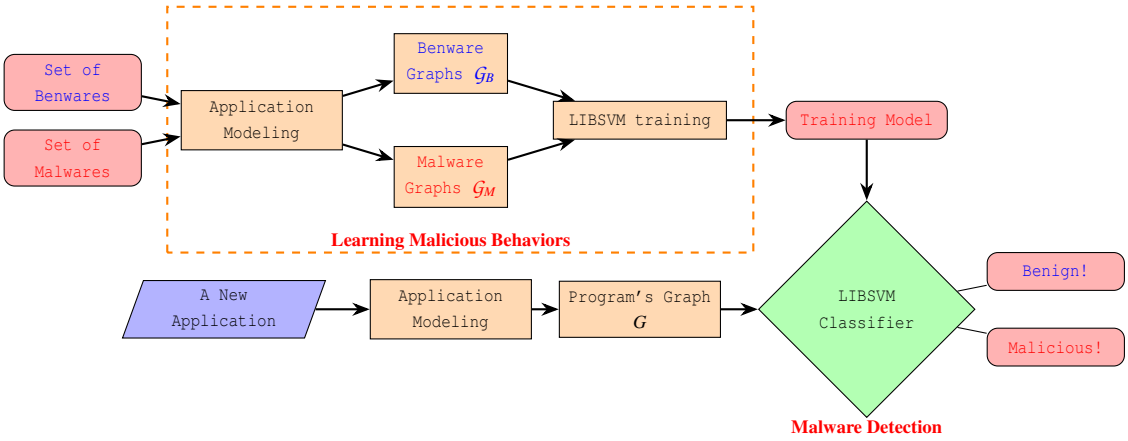

Figure 5: Learning Malicious Behaviors - Module 3.

Installing malicious packages. This malicious behavior installs a list of malicious packages to the system. It is shown in the malicious API graph of Figure 6. This graph represents the following malicious behavior: the method `get()` in the object `Map` is first called to get the package from the list. Then, the method `exec()` is called to execute the package and `getLaunchIntentForPackage()` is called to install the package. Finally, this package is removed from the list by calling the method `remove()` in the object `Map`.

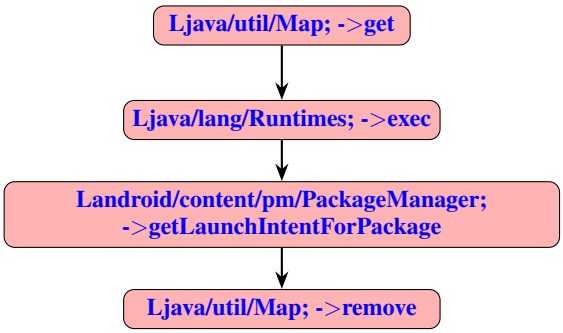

Figure 6: The malicious API graph of the Android malicious behavior of Installing malicious packages.

Running a malicious process. This malicious behavior replaces the running process by a malicious process. It is shown in the graph of Figure 7. The malicious behavior is implemented as follows: The ma-

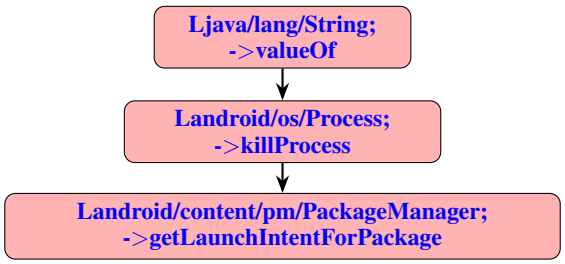

Figure 7: The malicious API graph of the Android malicious behavior of Running a malicious process.

licious application first calls `valueOf()` in the object `String` to get the identifier of the specific process. Then, it calls `killProcess()` to kill this process. Finally, it calls `getLaunchIntentForPackage()` to launch a replaced process in the system.

Repeatedly sending messages. This malicious behavior repeatedly sends the data via SMS messages. It is shown in the graph of Figure 8. The ma-

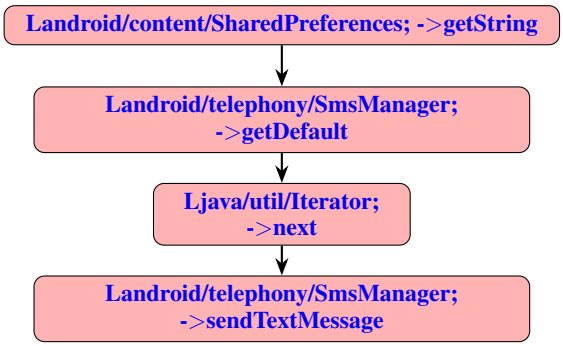

Figure 8: The malicious API graph of the Android malicious behavior of Repeatedly sending messages.

licious behavior is implemented as follows: The malicious application first calls `getString()` in `SharedPreferences` to get the values of the variables. Then, it calls `getDefault()` in `SmsManager` to get the object to handle SMS message in the system. Finally, it calls `sendTextMessage()` in `SmsManager` to send the messages. Besides, it calls `next()` in `Iterator` to get a list of messages for sending.

REFERENCES

Symantec (2016). Internet security threat report. https://www.symantec.com/security-center/threat-report. Accessed: 2016-11-25.

Apk Android Tool (2016). A tool for reverse engineering android apk files. https://ibotpeaches.github.io/Apktool. Accessed: 2016-11-25.

Kaspersky (2018). Internet security threat report. https://securelist.com/it-threat-evolution-q3-2017-statistics/83131/. Accessed: 2018-01-30.

McAfee (2019). Mcafee mobile threat report. https://www.mcafee.com/enterprise/en-us/assets/reports/rp-mobile-threat-report-2019.pdf. Accessed: 2020-09-30.

Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., and Rieck, K. (2014). Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*.

Burguera, I., Zurutuza, U., and Nadjm-Tehrani, S. (2011). Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*.

Canfora, G., Medvet, E., Mercaldo, F., and Visaggio, C. A. (2015). Detecting android malware using sequences of system calls. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*.

Chang, C.-C. and Lin, C.-J. (2011). Libsvm: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2. Software available at http://www.csie.ntu.edu.tw/ cjlin/libsvm.

Dam, K. and Touili, T. (2019a). STAMAD: a static malware detector. In *Proceedings of the 14th International Conference on Availability, Reliability and Security, ARES 2019, Canterbury, UK, August 26-29, 2019*, pages 25:1–25:6. ACM.

Dam, K.-H.-T. and Touili, T. (2017a). Extracting android malicious behaviors. In *International Workshop on FORmal methods for Security Engineering*.

Dam, K.-H.-T. and Touili, T. (2017b). Learning android malware. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ARES '17.

Dam, K. H. T. and Touili, T. (2019b). Stamad: A static malware detector. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, ARES '19, New York, NY, USA. Association for Computing Machinery.

Dimjaševic, M., Atzeni, S., Ugrina, I., and Rakamaric, Z. (2015). Android malware detection based on system calls. *University of Utah, Tech. Rep*.

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*, pages 45–54. ACM.

Jang, J.-w., Kang, H., Woo, J., Mohaisen, A., and Kim, H. K. (2016). Andro-dumpsys: anti-malware system based on the similarity of malware creator and malware centric information. *Computers & security*.

Maiorca, D., Ariu, D., Corona, I., Aresu, M., and Giacinto, G. (2015). Stealth attacks: An extended

insight into the obfuscation effects on android malware. *Computers & Security*.

Malik, S. and Khatter, K. (2016). System call analysis of android malware families. *Indian Journal of Science and Technology*.

Preda, M. D. and Maggi, F. (2016). Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology. *Journal of Computer Virology and Hacking Techniques*.

Rastogi, V., Chen, Y., and Jiang, X. (2013). Droidchameleon: Evaluating android anti-malware against transformation attacks. ASIA CCS '13.

Song, F. and Touili, T. (2014). Model-checking for android malware detection. In *Asian Symposium on Programming Languages and Systems*, pages 216–235. Springer.

Zheng, M., Lee, P. P. C., and Lui, J. C. S. (2013). *ADAM: An Automatic and Extensible Platform to Stress Test Android Anti-virus Systems*. DIMVA 2012.