Extracting Android Malicious Behaviors*

Khanh-Huu-The Dam[1] and Tayssir Touili[2]
[1]*University Paris Diderot and LIPN, Villetaneuse, France*
[2]*LIPN, CNRS and University Paris 13, Villetaneuse, France*

Keywords: Malware Detection, Android Malwares, Static Analysis, Information Retrieval.

Abstract: The number of Android malwares is increasing quickly. That makes the Android devices more vulnerable while they are the target of malware's writers. Thus, the challenge nowadays is to detect the malicious Android applications. To this aim, we need to know what are the malicious behaviors that Android malwares apply. In this paper, we introduce a method to *automatically* extract the malicious behaviors for Android malware detection. We present the behaviors of an Android application by an API call graph and we use a malicious API graph to represent the malicious behaviors. Then, given a set of malicious and benign applications, we compute the malicious behaviors by extracting from the API call graphs the subgraphs that are relevant to the malicious API call graphs but not relevant to the benign ones. This relevance is measured by applying the TFIDF weighting scheme widely used in the Information Retrieval Community. These malicious API graphs are applied to detect malicious applications. We obtained encouraging results with a recall rate of 92% and a precision of 98%.

1 INTRODUCTION

Since the number of Android users is growing very fast in recent years, the number of Android applications is also increasing as well. As a consequence of that growth, the target of malware's writers is changing and focusing on Android users. Thus, the number of variants of Android malwares increases year by year. According to the report of Symantec[2], this number increased from 3262 in 2013 to 3944 in 2015. Thus, the challenge is to detect the malicious Android applications.

A well known technique for Android malware detection is based on the analysis of the permission requirements. This technique consists of the analysis of the Android manifest file where all the required permissions and necessary components of an Android application are declared. These permissions are required by an application when users install it to their device. Then, once these permissions are gained for an application, it can make harmful operations to the system without informing to the user. So, the permission requirements can be a signature to distinguish the malicious applications. There are serval works (Aung and Zaw, 2013; Zhang et al., 2013; Tchakounté, 2014; Talha et al., 2015) that applied this technique to detect Android malwares. However, (Aafer et al., 2013) shows that detecting malwares by analyzing the permission requirements is not robust, especially when the malicious behavior can be executed without any additional permission. Moreover, in Android OS version 6.0 and later versions, the applications are able to request the permissions at run-time [3]. Thus, these permissions are not stored in the manifest file.

To overcome this limitation, other works (Burguera et al., 2011; Dimjaševic et al., 2015; Canfora et al., 2015; Jang et al., 2016; Malik and Khatter, 2016) try to detect Android malwares by dynamically analyzing the execution of the Android applications. In these works, the behaviors of an application is analyzed via its execution traces while running it in a simulated environment. However, the dynamic analysis allows only to analyze the behaviors of Android applications in a limited time interval. Thus, it cannot detect the malicious behavior if it occurs after this time interval.

To sidestep the limitations of the above approaches, (Aafer et al., 2013; Sharma and Dash, 2014; Song and Touili, 2014) apply static analysis for detecting the malicious applications. In these works, the authors specify the malicious behaviors by sequences of API calls of the Android application. APIs (Ap-

*This work was partially funded by the FUI project AIC 2.0.

[2]https://www.symantec.com/security-center/threat-report

[3]https://developer.android.com/guide/topics/security/permissions.html

Dam, K-H-T. and Touili, T.
Extracting Android Malicious Behaviors.
DOI: 10.5220/0006288807140723
In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy (ICISSP 2017)*, pages 714-723
ISBN: 978-989-758-209-7

plication Programming Interfaces) are functions supported by the Android OS and Android applications use them to access the system services and the system data. Obviously, the harmful tasks related to the system are operated by API calls. For instance, (Aafer et al., 2013; Sharma and Dash, 2014) analyze the applications by looking at the different calls to these API functions. In a more advanced analysis, (Song and Touili, 2014) specifies the behaviors (not the syntax) of the Android applications by doing the static analysis on the Android codes without installing/executing it. However, in this work the malicious behaviors are discovered by studying manually the Android codes. This task is time consuming and requires a huge engineering effort. Thus, one of the current challenges in Android malware detection is the automatic extraction of malicious behaviors.

In this paper, we propose a way to extract the malicious behavior of the Android applications *automatically*. Following (Aafer et al., 2013; Sharma and Dash, 2014; Song and Touili, 2014), we model the malicious behaviors via API calls. 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

```
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.

use a malicious API graph which is a graph whose nodes are API functions, and whose edges (f, f') express that API function f is called before API function f'. 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().

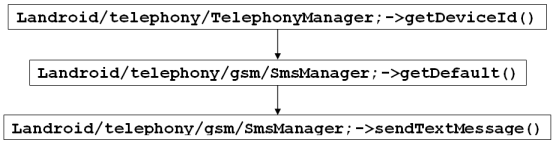


Figure 2: A malicious API graph of an Android trojan SMS spy.

Using this representation, the goal of this paper is to *automatically* extract such malicious API graphs. In order to do that, we represent the Android application using an API call graph, which is a graph whose nodes are pairs (m, f) consisting of an API f and a control point m, and whose edges $((m, f), (m', f'))$ express that there is a call to the API f at the control point m, followed by a call to the API f' at the control point m'. After extracting this malicious API graph, the malware detection phase is done by making a kind of product between the API call graph of the new application and the malicious API graphs. The application is marked as malicious if the product contains a feasible trace. Otherwise, it is marked as benign.

Then, given a set of API call graphs that correspond to malicious applications and a set of API call graphs corresponding to benign applications, we want to extract in a completely automatic way the malicious API graphs that correspond to the malicious behaviors of these malicious applications. The malicious API graph can be seen as a subgraph of the API call graphs of the malicious applications that represents the malicious behavior. The sufficient subgraphs that should be extracted can be used to distinguish the malicious API call graphs from the benign ones. Thus, our goal is to isolate the few relevant subgraphs from the irrelevant ones. This problem can be seen as an Information Retrieval (IR) problem, where the goal is to retrieve the relevant items and reject the irrelevant ones. The IR community have extensively studied this problem over the last 35 years. Several approaches have been proposed and successfully applied in several applications, such as email searching, text searching, image searching, etc. One of the most popular techniques in IR is the TFIDF weighting scheme where relevant terms are extracted from the documents in a collection by associating a weight to each term. The term that has the higher weight is the more relevant. In this paper, we adapt the TFIDF weighting scheme in IR to our API call graphs in or-

der to generate the malicious graphs. For that, we associate to each node and each edge in the API call graphs of the Android applications in the collection a weight. Then, the malicious API graphs are computed by taking edges and nodes with the highest weights.

We implemented this technique in a tool which consists of two phases: the training phase and the detection phase. In the training phase, we extract the malicious API graphs from the training set which is a collection of malicious applications and benign applications: First, we unpack and decompile each Android application from the Android package file (APK file). Then, we construct the control flow graph to represent the execution of this application. After that, we model its behaviors in an API call graph. Once we get the API call graphs of all the applications, we apply the TFIDF weighting scheme to get the relevant terms in the malicious graphs and automatically extract the malicious API graphs. In the detection phase, we use these malicious API graphs for detecting the malicious behaviors in the new applications. In order to evaluate the performance of our approach, we apply the tool on a data set of 3100 malicious applications which are collected from Drebin data set (Arp et al., 2014) and 459 benign applications taken from apkpure.com. We obtained encouraging results with a recall rate of 92% and a precision rate of 98%.

Outline. We present a way to construct the control flow graph for an Android application and define our API call graph model in Section 3. Section 4 introduces our malicious API graphs. In Section 5, we present the malicious behavior computation. Experiments are given in Section 6.

2 RELATED WORKS

Information retrieval techniques were applied for malware detection in (Masud et al., 2008; Cheng et al., 2013; Santos et al., 2013; Dam and Touili, 2016). . However, these works do not consider Android malwares. In this work, we extend the work of (Dam and Touili, 2016) to Android malwares.

The first technique used for Android malware detection is based on the analysis of the manifest file where the required permissions are stored. (Aung and Zaw, 2013; Talha et al., 2015) extract the set of permissions needed for malicious behaviors. (Aung and Zaw, 2013) constructs a bit-vector of permissions to represent android applications and implements a classifier to distinguish the malicious applications from the benign ones. As for (Talha et al., 2015), they extract information from the Android manifest file, such as application permissions, application services and application receivers to build a database of signatures for malware detection. As we have mentioned before, Android malware detection based on the analysis of permissions is not robust since the malicious behaviors may occur without any declaration in the Android manifest file.

API calls are used for Android malware specification in (Aafer et al., 2013; Arp et al., 2014; Sharma and Dash, 2014; Jang et al., 2016). (Sharma and Dash, 2014; Arp et al., 2014) analysze the manifest file and the API calls to specify the behaviors. They filter out the suspicious APIs which are potentially used in the malicious behaviors. In (Sharma and Dash, 2014), the authors select 35 features from the set of features for classifying the malicious applications. (Arp et al., 2014) construct a huge bit-vector of roughly 545,000 features to classify malicious applications.

(Aafer et al., 2013) use API calls to describe behaviors of an application and use common classifiers such as ID3, SVM and C4.5 to classify the malicious behaviors. (Jang et al., 2016) generates a profile of the API calls that are invoked in the application to specify the behavior of the application. This profile describes the usages of all API functions and their objects in an application. The authors introduce a decision process to detect malicious applications by comparing the similarities between profiles.

In another work, (Canfora et al., 2015) considers sequences of system calls as a specification of the application's behaviors. The occurrences of subsequences with length n in each sequence is taken into account in the construction of the feature vector to represent the applications. Then, the authors implement a SVM classifier for malware detection. However, they use dynamic analysis to capture the traces of the application's execution. Thus, they take into account only *one* execution of the application.

In our work, we also use API calls to represent the behaviors of an Android application. However, with the API call graph representation we take into account the order of API function calls in all the executions of an application. So, our API call graphs are more precise than the representations used in the works cited above. Moreover, we are able to extract the malicious behaviors of the malicious applications as malicious API graphs while none of the above works extract the malicious behaviors of Android applications automatically. Our malicious API graphs are also used to detect malicious applications. Furthermore, our technique is completely static, we do not use any dynamic analysis.

(Song and Touili, 2014) apply static analysis for Android malware detection. However, this work is based on manually studying the codes of the Android

applications to extract the malicious behaviors while our work allows to extract automatically these behaviors.

3 MODELING ANDROID APPLICATIONS

During the execution of an Android application, different events from the users or the system need to be taken into account. Thus, the entry points of an Android application are specified by the event handlers which are methods called when there is an event that occurs in the system or from the user like opening the application, touch on the screen, etc. By taking into account the event handlers in the Android code, we construct the control flow graph which represents all the operations in an Android application. Then, we extract an API call graph to represent the behaviors of the application. We introduce all these steps in this section.

3.1 Android Applications

An Android application is installed into the Android devices via an Android package (an APK file) which consists of (1) Dalvik bytecode (a dex file) which is executable, (2) all resources, and (3) an Android manifest file which contains the declaration of all the permission requirements and necessary components used in the application's execution. In order to analyze the behaviors of an Android application, we have to look into the Android byte code which is executed whenever the application runs. However, this code is a kind of binary code for Android programs. It is hard for humans to read. Thus, we use a tool to decompile this byte code into a human readable code, called smali code. Thanks to the Apktool[4], the Android package is unpacked and the byte code is decompiled into the smali code. The smali code represents the Android program in a form of an object oriented programming language (like Java language) where a program is constructed from objects that contain data and functions known as methods. When the Android application runs, one of its objects is executed and in this executed object there are calls to other objects.

An Android application is implemented mainly using four main components: Activity, Service, Broadcast Receiver and Content Provider. Each component supports different functions in the application. For instance, the component Activity provides a user interface for interactions between the user and the application. Based on the component Service, the developer can make background executions. The component Broadcast Receiver is a means to make communications between applications in the system. Via the Content Provider objects, the application can access/modify data on the system such as Contact list, Calendar, etc. Each component follows its own work flow. For example, in the component Activity, first the method onCreate() is called, then the method onStart(), etc. These components contain the entry points to execute the application. Thus, they are potentially exploited for malicious behaviors.

As an example, Figure 1 shows the implementation of an Android application based on the component Activity. The application starts by executing the object MainActivity which is an implementation of the component Activity. This object starts by calling the method onCreate() and then calling the method onStart(). Particularly, in the method onStart(), there is a call to the method Send() at line 13 which belongs to another object (the SendMessage object).

Moreover, there are several calls to the methods which are supported by the Android OS such as the method getDeviceId() at line 5 in the object TelephonyManager, the method getDefault() at line 19 and the method sendTextMessage() at line 25 in the object SmsManager. These methods are functions which are provided by the Android OS to access the system services. Each Android OS version supports a library which includes all functions to implement an application, called Application Programming Interfaces (APIs). The Android developers use these APIs to create the application based on the above components. Generally, the system operations are made via API calls. The malicious behaviors in an Android application are used to be done by sequences of API calls. According to this fact, we represent the behaviors of an Android application as an API call graph which represents all sequences of API calls. In order to construct the behaviors of an Android application, we build a control flow graph which represents all operations in the application. Then, we extract the API call graph from the control flow graph. The API call graph is seen as the representation of the behaviors of an Android application. In the following subsections, we introduce control flow graphs and API call graphs, and explain how to compute them for Android applications.

3.2 Control Flow Graph

A control flow graph (CFG) is a directed graph $G = (N, I, E)$, where N is a finite set of nodes, I is a finite set of instructions in an Android application, and

[4]https://ibotpeaches.github.io/Apktool

$E : N \times I \times N$ is a finite set of edges. Each node corresponds to a control point in the Android application. Each edge specifies the connection of two control points and is associated with an instruction. An edge (n_1, i, n_2) in E expresses that in the Android application, the control point n_1 is followed by the control point n_2 and is associated with the instruction i.

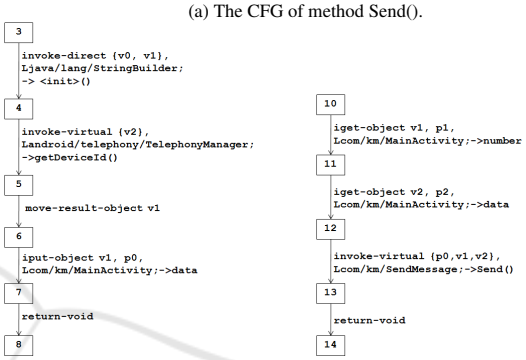
We construct the CFG of an Android application as follows:

1. Build the CFG for each method in the application.
2. If at a given control point n there exists a call to another method A $(n \xrightarrow{callA} n' \in E)$, we remove this edges from E and we make a link from point n where the call occurs to the entry point of the called method A and a link from the exit point of the called method A to the next point n' of this call.
3. By taking into account the event handlers of an Android application, we make the link between the CFGs of the methods which handle the events according to the work flow of each object. For instance, to handle the initialization of a new activity object, there is a call to the method onCreate() and then a call to the method onStart(). Thus, we add a link between the exit points of the method onCreate() to the entry points of the method onStart() in this object.

As an example, let us consider the smali code in Figure 1. Firstly, we build the CFG for each method as shown in Figure 3. Then, since there is a call to the method Send() from the method onStart(), we make a link from the point (12) where a call to the method Send() will be made to the entry point (17) of the method Send() and a link from the exit point (26) of the method Send() to the next point (13) of this call. This is shown in Figure 4.

Moreover, in this application there is an activity object. This object is started by calling the method onCreate() and then calling the method onStart(). Thus, we make a link from the exit point (8) of the method onCreate() to the entry point (10) of the method onStart(). The CFG of the application is shown in Figure 5.

(a) The CFG of method Send().



(b) The CFG of method onCreate(). (c) The CFG of method onStart().

Figure 3: The CFGs of the methods of the code in Figure 1.

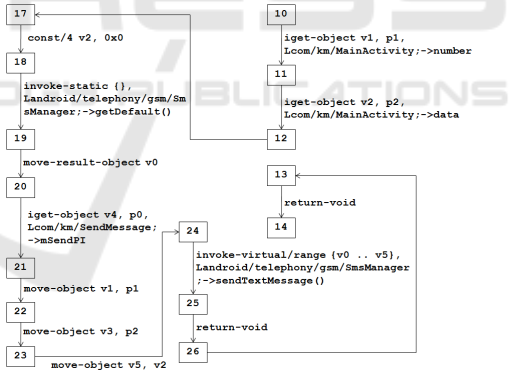


Figure 4: The CFG of method onStart() linked to the CFG of method Send().

3.3 API Call Graph

Let $\mathcal{A}$ be the set of all APIs in Android applications. An API call graph is a directed graph $G_{api} = (V_{api}, E_{api})$, where $V_{api} : N \times \mathcal{A}$ is a finite set of vertices and $E_{api} : (N \times \mathcal{A}) \times (N \times \mathcal{A})$ is a finite set of edges. A vertex (n, f) means that at a control point n, a call to the API function f is made. An edge $((n_1, f_1), (n_2, f_2))$ in E means that the API function f_2 called at the control point n_2 is executed after the API function f_1 called at the control point n_1. To compute an API call graph from the CFG of an Android application, we perform a kind of control point reachability analysis on the CFG as described in (Dam and Touili, 2016).

As an example, let us consider the construction of the API call graph of the code in Figure 1. Since the control flow graph is constructed in Figure 5, we simplify this graph to get an API call graph where each node is a pair of the control point and the API which is called at this control point. Then, we make

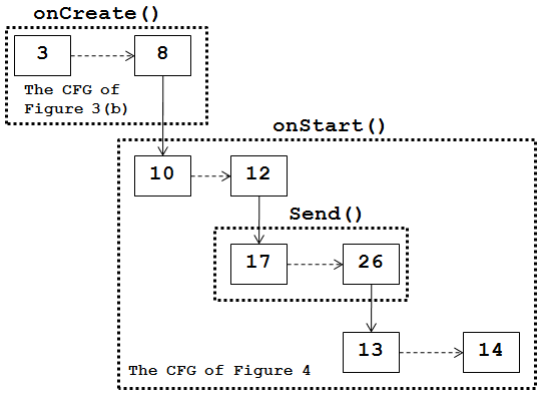


Figure 5: The CFG of the code in Figure 1.

the transitive closure for each node in this simplified graph. The API call graph is shown in Figure 6.

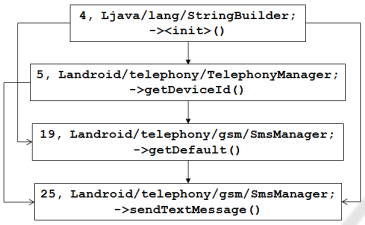


Figure 6: The API call graph of the code in Figure 1.

4 MALICIOUS BEHAVIOR SPECIFICATION

In this section we introduce *malicious API graphs*, and we show how to apply these graphs to detect malwares.

4.1 Malicious API Graphs

A *malicious API graph* is a directed graph $G_M = (V_M, E_M, V_0, V_F)$, where $V_M \subseteq \mathcal{A}$ is a finite set of vertices, $E_M : V_M \times V_M$ is a finite set of edges, $V_0 \subseteq V_M$ is the set of initial nodes, i.e., the set of nodes v s.t. there does not exist any edge coming to v, and $V_F \subseteq V_M$ is the set of final nodes, i.e., the set of nodes v s.t. there does not exist any edge exiting v. An edge (f_1, f_2) in E_M means that the API function f_2 is executed after the API function f_1. Let $f_0, \ldots, f_k$ be API functions. A malicious behavior $f_0 \cdots f_k$ is represented in the malicious API graph if $f_0 \in V_0$, $f_k \in V_F$, and for every $i, 1 \leq i \leq k-1$, $(f_i, f_{i+1}) \in E_M$.

For instance, the malicious behavior of the Android trojan sms spy is expressed by the malicious API graph of Figure 2. In this graph, the behavior starts at the initial node Landroid/telephony/TelephonyManager; -¿ getDeviceId() and ends at the final node Landroid/telephony/gsm/SmsManager; -¿ sendTextMessage(). This graph represents the behavior of getting and sending the Phone Id via a text message.

4.2 Malware Detection using Malicious API Graphs

Given a program represented by its API call graph $G_{api} = (V_{api}, E_{api})$, and a set $\mathcal{M}$ of malicious behaviors (sequences of API functions) represented by a malicious API graph $G_M = (V_M, E_M, V_0, V_F)$, we check whether the program contains one of the malicious behaviors in $\mathcal{M}$, by performing a kind of product as follows: $G_P = (V_P, E_P, V_0^P, V_F^P)$ such that $V_0^P = \{(m,f) \in V_{api} \mid f \in V_0\}$, $V_F^P = \{(m,f) \in V_{api} \mid f \in V_F\}$, and $E_P = \{\big((m,f),(m',f')\big) \in E_{api} \mid (f,f') \in E_M\}$. Then, the program contains a malicious behavior in $\mathcal{M}$ iff G_P contains paths that led from an initial node in V_0^P to a final node in V_F^P.

5 MALICIOUS BEHAVIORS EXTRACTION

The problem is to compute the malicious API graph from a set of malicious and benign API call graphs. Following the work in (Dam and Touili, 2016), we compute 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 meaningful to the malicious API call graphs. Applying the techniques of the information retrieval community, we associate to each node/edge of the API call graphs a weight to measure its relevance with respect to the malicious graphs and wrt. the benign ones. Using these weights, we construct the malicious API graphs. In this section, we recall the approach of (Dam and Touili, 2016) and show how to apply the TFIDF scheme that was widely applied for documents by the IR community for our malicious graph extraction problem.

5.1 Term Weighting Scheme

Relevance in a Graph. In what follows, we call nodes and edges terms. The relevance of a term is measured by a TFIDF weight. This scheme ensures that if a term has a higher weight, then it is more relevant in the graph. The TFIDF weighting scheme is a well-known technique of the IR community that is applied in many applications in web searching, text

searching, image searching, etc. It was mainly applied for documents. Following (Dam and Touili, 2016), we show here how to apply it for graphs. The weight in the TFIDF scheme is measured from the occurrences of terms in a graph and their appearances in other graphs. If a term occurs frequently in a graph and appears rarely in other graphs, it is one of the relevant terms in this graph. For a given term, its relevance to an API call graphs in the collection $\mathcal{G}$ is measured by the TFIDF weight as follows.

$$w(i,j) = F(\mathbf{tf}(i,j)) \times \mathbf{idf}(i) \quad (1)$$

where $w(i,j)$ is the weight of a term i in graph j. The **idf** factor ensures a higher weight for terms which appear in a few graphs of the collection. It varies inversely with the number of graphs $\mathbf{df}(i)$ that contain a term i in a collection of N graphs. A typical factor may be computed as $\log(\frac{N}{\mathbf{df}(i)})$. $\mathbf{tf}(i,j)$ is the number of occurrences of term i in graph j, called term frequency.

Moreover, a term in a large-size graph may have a high term frequency. If in a collection the difference between the sizes of graphs is high, the term frequencies are taken over by the large-size graphs. Thus, one needs to take into account the size of the graphs while computing the term frequency. This is implemented using the function F that involves a size normalization component (Robertson et al., 1995; Singhal et al., 1996). In our experiment, we apply several functions of term frequency. They are defined as follows:

- $F_1(\mathbf{tf}(i,j)) = \mathbf{tf}(i,j)$ leads to the raw factor.
- $F_2(\mathbf{tf}(i,j)) = \frac{(k_1+1)\times\mathbf{tf}(i,j)}{\mathbf{tf}(i,j)+k_1(\frac{\mathbf{S}(j)}{\mathbf{AVG}(\mathcal{G})}\times b+1-b)}$ implements the size normalized BM25 factor (Robertson and Zaragoza, 2009; Robertson et al., 1995).
- The size normalized logarithmic factor can be implemented by the following function (Singhal et al., 1999; Singhal and Kaszkiel, 2001):

$$F_3(\mathbf{tf}(i,j)) = \begin{cases} \frac{1+ln(1+ln(\mathbf{tf}(i,j)))}{\frac{\mathbf{S}(j)}{\mathbf{AVG}(\mathcal{G})}\times b+1-b} & \text{if } \mathbf{tf}(i,j) > 0 \\ 0 & \text{if } \mathbf{tf}(i,j) = 0 \end{cases}$$

- The size normalized sigmoid factor (Yao et al., 2006) can be implemented by function F_4 defined as follows:

$$\begin{cases} \frac{k_1+1}{k_1(\frac{\mathbf{S}(j)}{\mathbf{AVG}(\mathcal{G})}\times b+1-b)+e^{-\mathbf{tf}(i,j)}} & \text{if } \mathbf{tf}(i,j) > 0 \\ 0 & \text{if } \mathbf{tf}(i,j) = 0 \end{cases}$$

Where $\frac{\mathbf{S}(j)}{\mathbf{AVG}(\mathcal{G})} \times b + (1-b), 0 \leq b \leq 1$ is a size normalization component (Singhal et al., 1996) where $\mathbf{S}(j)$ is the size of graph j and $\mathbf{AVG}(\mathcal{G})$ is the average size of graphs in the collection $\mathcal{G}$. In the above formulas, by setting b to 1, graph size normalization is fully performed, while setting b to 0 turns off the size normalization effect.

In the TFIDF weighting scheme, these functions are applied and have shown good performances. However, there is no theoretical evidences to prove which function is better than the others. Thus, we take into account all these functions in our experiments to determine which one is the best function for our problem.

Relevance in a Set. Given a set $\mathcal{G}$ of API call graphs and a term i, the relevance of term i in $\mathcal{G}$ is measured by its relevance in each graph in this set: it is computed as the sum of the term weights of i in each graph of $\mathcal{G}$:

$$W(i,\mathcal{G}) = \frac{1}{K}\sum_{j=1}^{|\mathcal{G}|} w(i,j) \quad (2)$$

where $K = \max_{i,j} w(i,j)$ is a normalizing coefficient. It is used to normalize the term weight values in the different graphs to make them comparable in the summation. $w(i,j)$ is a TFIDF term weight of term i in graph j ($j \in \mathcal{G}$). $w(i,j)$ is computed using one of the functions F as described above. $W(i,\mathcal{G})$ is the weight of term i in the set $\mathcal{G}$. A term with a higher weight is more relevant to graphs in set $\mathcal{G}$.

Malicious Relevance. In our context, given $\mathcal{G}_M$ and $\mathcal{G}_B$ that are the sets of malicious graphs and benign graphs respectively, we want to compute a term weight that is high if the term is relevant for set $\mathcal{G}_M$ but not for set $\mathcal{G}_B$. If a term i is relevant in both sets, then it is not meaningful, as it does not correspond to a malicious behavior. Thus, given a term i we need to compute a new weight of term i to measure its relevance in set $\mathcal{G}_M$ with respect to set $\mathcal{G}_B$. This is computed by two intuitive equations.

The first equation is the Rocchio equation which intuitively measures the weight of a term by the distance between its weights in the sets $\mathcal{G}_M$ and $\mathcal{G}_B$ (Christopher D. Manning, 2009). A higher distance of a term means that it is more relevant for $\mathcal{G}_M$ than for $\mathcal{G}_B$. Given a term i, the relevance of term i in the set $\mathcal{G}_M$ against the other set $\mathcal{G}_B$ is given by:

$$W(i,\mathcal{G}_M,\mathcal{G}_B) = \beta \times \frac{W(i,\mathcal{G}_M)}{|\mathcal{G}_M|} - \gamma \times \frac{W(i,\mathcal{G}_B)}{|\mathcal{G}_B|} \quad (3)$$

where $|\mathcal{G}_M|$ and $|\mathcal{G}_B|$ are the sizes of the sets $\mathcal{G}_M$ and $\mathcal{G}_B$, β and γ are parameters to control the effect of the two sets $\mathcal{G}_M$ and $\mathcal{G}_B$.

The second equation is the Ratio equation which computes the relevance of a term i as follows:

$$W(i,\mathcal{G}_M,\mathcal{G}_B) = \frac{W(i,\mathcal{G}_M)}{|\mathcal{G}_M|} \times \frac{\lambda + |\mathcal{G}_B|}{\lambda + W(i,\mathcal{G}_B)} \quad (4)$$

Intuitively, this is a kind of quotient between the weight of i in $\mathcal{G}_M$ and its weight in $\mathcal{G}_B$. Thus, the weight of term i is high if it has a high weight in $\mathcal{G}_M$ and a low weight in $\mathcal{G}_B$.

As the two equations above (Rocchio and Ratio) are natural and intuitive, a higher weight indicates a higher relevance of a term i in $\mathcal{G}_M$. We have no theoretical evidence that shows the advantage of taking one equation over the other. Thus, in our experiments, we make a comparison of the performances of these two equations in order to decide which one is the best for our application. In our experiments, the values of β, γ and λ are set to $0.15, 0.75$ and 0.5, respectively. (These are the typical values used in the IR community.)

5.2 Computing Malicious API Graphs

A malicious API graph is a combination of terms which have a high malicious relevance. Since terms of a graph are either nodes or edges, we can compute the malicious API graphs by several possible strategies. These strategies depend on a parameter n which is chosen by the user. n corresponds to the number of nodes (resp. edges) that are taken into account in the computation. In what follows, by "weight of a term i", we refer to $W(i, \mathcal{G}_M, \mathcal{G}_B)$. Let $\{(m_1, f_1), \ldots, (m_n, f_n)\}$ be the set of nodes that have the n highest weights. Let $\mathcal{A}_\mathcal{M} = \{f_1, \ldots, f_n\} \subseteq \mathcal{A}$ be the corresponding API functions. Let $\{e_1, \ldots, e_n\}$ be the set of edges that have the n highest weights. Intuitively, this means that the API functions in $\mathcal{A}_\mathcal{M}$ and the edges in $\{e_1, \ldots, e_n\}$ are the most relevant ones. Then, the malicious API graphs are computed using the following strategies:

Strategy 1. We take nodes which correspond the API functions in $\mathcal{A}_\mathcal{M}$ and then add edges with the highest weight to connect these nodes. No other node is added.

Strategy 2. We start from nodes which correspond to the API functions $\mathcal{A}_\mathcal{M}$. For every node, we consider all its outgoing edges, and we add the edge with the highest weight, even if it involves a node that is not in $\mathcal{A}_\mathcal{M}$.

Strategy 3. We construct the graph from the edges with the highest weights $\{e_1, \ldots, e_n\}$.

6 EXPERIMENTS

We evaluate the performance of our approach on a data set of 459 benign applications, which are collected from the website apkpure.com and 3100 malicious applications which are gotten from Drebin dataset (Arp et al., 2014). We divide this data set into two sets for training and testing. The training set consists of 1900 malicious applications and 359 benign applications. The test set consists of 1200 malicious applications and 100 benign applications. Figure 7 shows the proportions of each malware category in the training set and in the test set. There are two phases in this evaluation as follows.

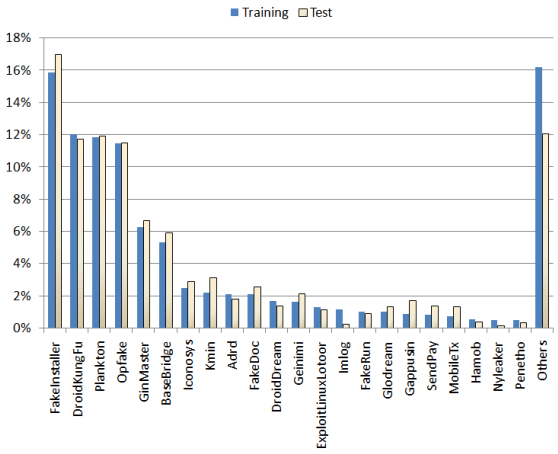


Figure 7: Proportions of malicious application categories in training and test sets.

- **Training Phase.** In this phase, we extract the malicious API graphs from the data in the training set as follows. Firstly, we take each input application as an APK file. Then, this APK file is unpacked and decompiled by Apktool. After getting its smali code from Apktool, we construct the CFG and the API call graph of this application. Once we build the API call graph for each application, we compute the malicious API graphs by the strategies described in Section 5.2.
- **Testing Phase.** In this phase, we apply the malicious API graphs to classify the applications in the test set. We take each input application as an APK file. Then, we unpack and decompile this APK file to get its smali code by using Apktool. After that, we construct its API call graph. As we described in Section 4.2, this application is marked as malicious if there is a feasible common path between its API call graph and the malicious API graphs. Otherwise, it is marked as benign.

We evaluate the performance of the functions F_1, F_2, F_3 and F_4 (detailed in Section 5.1) with the Rocchio and Ratio equations (equations 3 and 4). For each combination, we construct the malicious API graph from the training set. Then, these specifications are evaluated on the test set. The performance is measured by the following quantities. Recall (Detection rate) is the True Positives over the number of malicious API call graphs. Precision is the True

Table 1: The best performance of each strategy.

Strategy	n	Recall	Precision	F-measure
S1 by F4 with Rocchio equation	100	82.8%	97.3%	89.5%
S1 by F2 with Ratio equation	80	85.4%	98.7%	91.6%
S2 by formula F4 with Rocchio equation	100	87.6%	97.5%	92.3%
S2 by formula F2 with Ratio equation	100	88%	98.8%	93.1%
S3 by formula F3 with Rocchio equation	85	92%	98%	94.9%
S3 by formula F3 with Ratio equation	85	92%	98%	94.9%

Positives over the sum of True Positives and False Positives. F-Measure is a harmonic mean of precision and recall that is computed as F-Measure $= 2\times\text{Precision}\times\text{Recall}/(\text{Precision}+\text{Recall})$. These are standard evaluation measures in the IR community. The technique computes more relevant items than irrelevant if the precision rate is high. We can deduce that most of the relevant items were computed if the recall rate is high. As for the F-measure, it is 1 if all retrieved items are relevant and all relevant items have been retrieved.

The figures below show the F-measure obtained from the combinations of different formulas and strategies. In these experiments, we vary n from 5 to 120. According to the results, the ratio equation gives the best performance with formulas F2 and F3 while the Rocchio equation gives the best performance with formula F4. Moreover, the F-measure is more stable (no jumps) with the ratio equation than with the Rocchio equation. Table 1 shows the best obtained results for each combination. We obtain the best performance from the strategy S3 by F3 with both weighting equations. The detection rate reaches 92% with 98% of precision at $n = 85$. Therefore, we use this configuration (S3 by F3 with the ratio equation) for malicious graphs extraction because the ratio equation is more stable.

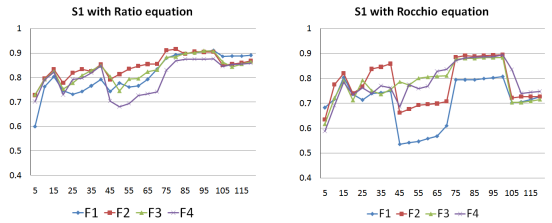


Figure 8: F-measures of strategy S1 on the test set.

Morever, our tool was able to *automatically* extract several graphs that represent real malicious behaviors. For example, it was able to extract the graph of Figure 11 below.
This behavior consists of repeatedly sending the user information by text messages. First, the method createFromPdu() is called to create a new text message. Then, the method getUserData() is called to get the information of the user. Additionally, more information is gotten from calling getSharedPreferences(). The message is sent by calling sendTextMessage(). This process is repeated by an iterator object.

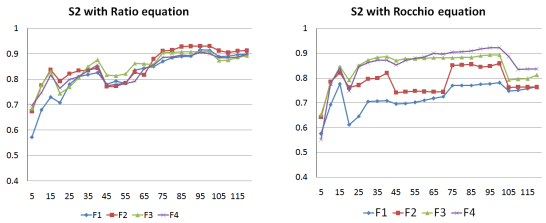


Figure 9: F-measures of strategy S2 on the test set.

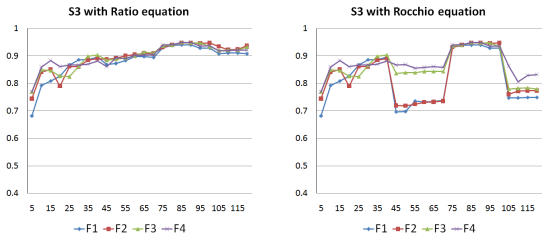


Figure 10: F-measures of strategy S3 on the test set.

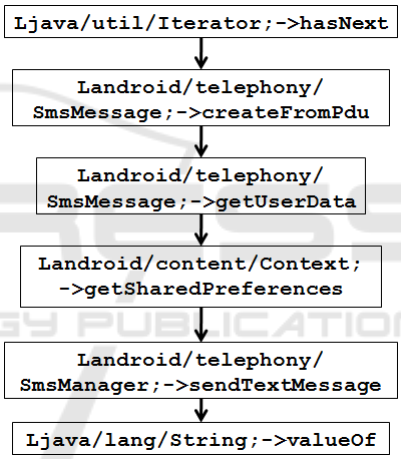


Figure 11: Repeatedly sending text messages.

7 CONCLUSION

In this paper, we consider the problem of automatically extracting Android malicious behaviors. To solve this problem, we consider a set of benign and malicious Android applications. We model these applications using API call graphs, and we extract from these graphs the relevant subgraphs that form the malicious specifications by using and adapting well known techniques in the Information Retrieval community. Using the graphs generated by our techniques, we obtained interesting results: 92% of detection rates with 98% of precision. As far as we know, this is the first time that information retrieval

techniques are applied for the automatic extraction of Android malicious behaviors.

REFERENCES

Aafer, Y., Du, W., and Yin, H. (2013). Droidapiminer: Mining api-level features for robust malware detection in android. In *International Conference on Security and Privacy in Communication Systems*, pages 86–103. Springer.

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*.

Aung, Z. and Zaw, W. (2013). Permission-based android malware detection. *International Journal of Scientific and Technology Research*, 2(3):228–234.

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*, pages 15–26. ACM.

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*, pages 13–20. ACM.

Cheng, J. Y.-C., Tsai, T.-S., and Yang, C.-S. (2013). An information retrieval approach for malware classification based on windows api calls. In *2013 International Conference on Machine Learning and Cybernetics*.

Christopher D. Manning, Prabhakar Raghavan, H. S. (2009). An introduction to information retrieval. Cambridge University Press.

Dam, K.-H.-T. and Touili, T. (2016). Automatic extraction of malicious behaviors. In *11th International Conference on Malicious and Unwanted Software 2016 (MALCON 2016)*, Fajardo, Puerto Rico.

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

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*, 58:125–138.

Malik, S. and Khatter, K. (2016). System call analysis of android malware families. *Indian Journal of Science and Technology*, 9(21).

Masud, M. M., Khan, L., and Thuraisingham, B. (2008). A scalable multi-level feature extraction technique to detect malicious executables. *Information Systems Frontiers*.

Robertson, S. and Zaragoza, H. (2009). *The probabilistic relevance framework: BM25 and beyond*. Now Publishers Inc.

Robertson, S. E., Walker, S., Jones, S., Hancock-Beaulieu, M. M., Gatford, M., et al. (1995). Okapi at trec-3. *NIST SPECIAL PUBLICATION SP*.

Santos, I., Ugarte-Pedrero, X., Brezo, F., Bringas, P. G., and Gómez-Hidalgo, J. M. (2013). Noa: An information retrieval based malware detection system. *Computing and Informatics*.

Sharma, A. and Dash, S. K. (2014). Mining api calls and permissions for android malware detection. In *International Conference on Cryptology and Network Security*, pages 191–205. Springer.

Singhal, A., Buckley, C., and Mitra, M. (1996). Pivoted document length normalization. In *Proceedings of the 19th annual international ACM SIGIR conference on Research and development in information retrieval*.

Singhal, A., Choi, J., Hindle, D., Lewis, D. D., and Pereira, F. (1999). At&t at trec-7. *NIST SPECIAL PUBLICATION SP*.

Singhal, A. and Kaszkiel, M. (2001). A case study in web search using trec algorithms. In *Proceedings of the 10th international conference on World Wide Web*.

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

Talha, K. A., Alper, D. I., and Aydin, C. (2015). {APK} auditor: Permission-based android malware detection system. *Digital Investigation*, 13:1 – 14.

Tchakounté, F. (2014). Permission-based malware detection mechanisms on android: Analysis and perspectives. *JOURNAL OF COMPUTER SCIENCE*, 1(2).

Yao, J., Wang, J., Li, Z., Li, M., and Ma, W.-Y. (2006). Ranking web news via homepage visual layout and cross-site voting. In *European Conference on Information Retrieval*.

Zhang, Y., Yang, M., Xu, B., Yang, Z., Gu, G., Ning, P., Wang, X. S., and Zang, B. (2013). Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 611–622, New York, NY, USA. ACM.