Automated static analysis and classification of Android malware using permission and API calls models

Anastasia Skovoroda
Lomonosov Moscow State University
Moscow, Russia
Email: nastya_jane@seclab.cs.msu.su

Dennis Gamayunov
Lomonosov Moscow State University
Moscow, Russia
Email: gamajun@seclab.cs.msu.su

***Abstract*—In this paper we propose a heuristic approach to static analysis of Android applications based on matching suspicious applications with the predefined malware models. Static models are built from Android capabilities and Android Framework API call chains used by the application. All of the analysis steps and model construction are fully automated. Therefore, the method can be easily deployed as one of the automated checks provided by mobile application marketplaces or other interested organizations.**

Using the proposed method we analyzed the Drebin and ISCX malware collections in order to find possible relationships and dependencies between samples in collections, and a large fraction of Google Play apps collected between 2013 and 2016 representing benign data. Analysis results show that a combination of relatively simple static features represented by permissions and API call chains is enough to perform binary classification between malware and benign apps, and even find the corresponding malware family, with an appropriate false positive rate of about 3%. Malware collections exploration results show that Android malware rarely uses obfuscation or encryption techniques to make static analysis more difficult, which is quite the opposite of what we see in the case of the 'Wintel' endpoint platform family.

We also provide the experiment-based comparison with the previously proposed state-of-the-art Android malware detection method *adagio*.

I. Introduction

Since mobile devices are currently used by the majority of the global population on a daily basis, the problem of mobile security has become essential for personal and corporate security. Most users store personal or confidential data on their devices including bank information, email, social networks, corporate applications and various authorization data. Obtaining users' private data is the goal of most mobile malware applications.

Many mitigation approaches have been proposed recently to detect and prevent mobile malware [1]. There are two major types of malware detection methods: anomaly detection and misuse detection. Misuse detection methods can be divided into two groups: methods utilizing some generic features of the malicious applications, and methods based on signatures/models of the known malicious samples. For example, taint-analysis aimed at detecting data leaks [2; 3] may be attributed to the first group. Detection capabilities of the 'generic feature' methods are limited to the malicious applications families which implement the corresponding generic feature. Specifying an algorithm to detect arbitrary malicious functionality is quite difficult and therefore such methods are usually limited to a narrow class of detected apps. Signature based methods have another drawback - they essentially depend on the utilized signatures/models and are not applicable to zero-day malware detection. Anomaly-based detection methods [4; 5] require specifying models of legitimate functionality which is even more difficult as it requires much wider code (or functionality) coverage compared to malicious functionality detection.

Based on the underlying techniques, detection methods may be divided into two large groups: static and dynamic analysis, depending on the types of data analyzed - static analysis operates on application code, and dynamic analysis operates on the actual execution of the analyzed application, in a native, virtual or emulated environment. More detailed classification of the detection methods may include machine learning methods, permission analysis, battery-life monitoring and cloud computing [1].

In this article we propose an automatic static analysis method for mobile applications using malicious application models. Application models reflect their behaviour and are built based on the used system permissions and API calls. Opposite to many other static analysis methods [6; 7] malware model construction does not require manual work on the part of the analyst. Analysis results represent the information on whether the similar model was found among malware models and the list of corresponding similar models. The method may be used by mobile application marketplaces, corporations with BYOD practices, or individual security researchers to analyze new applications before exposing them to users. It can also be used by anti-virus applications as a part of the complex analysis in the cloud.

We tested our method in binary classification experiment on Drebin and ISCX malware collections and a part of our collection of benign applications randomly choosing malicious application to generate models. The results of this experiment reveal the true negative rate of our method of about 91% and

the false positive rate of about 2%. We conducted the same experiment on the state-of-the-art Android malware detection method *adagio* and revealed that *adagio* has a wider detection coverage (true negative rate) while at the same time generating much more false alarms.

We also applied the proposed method to explore Drebin and ISCX malware datasets. We performed classification of malicious applications and automated grouping into families of similar applications using our method. One application in each family is later used to build a malware model for the family. The resulting clusters do not directly match the predefined malware families provided with the datasets which we believe to be obtained using manual analysis of each sample. As observed, apps inside predefined malware families may vary dramatically and for some of them it is very difficult to find anything common even manually. Therefore, we consider the proposed method to be a useful tool for detecting similarity between malware samples representing the same malware family. The paper also provides some observations of known malicious application features obtained during manual and automatic analysis of malware datasets.

Our evaluation of the method on a large number of benign applications revealed false positive rate of 3.2% which is acceptable for static analysis but probably too high to use the method independently of the other detection methods. If used separately, the information about found common API call chains gives malware analysts an easy way to cut off all the generated false alarms. Therefore, the method is very useful to simplify and automate mobile malware analysts' workflow. Considering automated usage, the method can be easily complemented with other detection approaches to get rid of the false positives. [1]

II. Related work

There is a lot of recently published research which addresses the problem of mobile malware, many of the proposed solutions seem very promising. Tam et al. give a comprehensive survey of mobile malware detection approaches alongside with the mobile malware evolution and its means of analysis evasion [8].

The proposed method is not unique in basing the analysis on API calls and Android permissions and building function call graph as a part of analysis. There are many machine learning methods using the feature set mainly composed of such application features [9; 10; 11; 12; 13; 14]. All of the listed methods, however, form their application models (feature vectors) based on rather simple application analysis and mostly rely upon well-suited to large-scale experiments machine learning algorithms.

MaMaDroid [9] builds behavioral models in the form of Markov chains from a sequence of abstracted API calls to extract application features and perform binary classification. MaMaDroid achieves 99% F-measure [2] when tested on a dataset with the applications not newer than the ones used for training. The authors also assess the impact of training dataset outdating: F-measure decreases to 75% for apps 2 years newer than the ones used for training and drops to 51% after 4 years.

Drebin [11] method utilizes SVM for malware classification. It uses a broad range of features: obtained statically hardware components requested by the app, the requested permisssions, filtered intents, app components, suspicious and restricted API calls, network addresses. The analysis also includes explanation in case the app is considered malicious. Drebin achieved false negative rate of 4.1–6.1% with false positive rate of 1% in its experimental evaluation.

Aafer et al. [12] compared API used by malicious and benign Android applications and revealed characteristic API with some conditions on parameters. These API calls are encountered in malicious applications at least 6% more often than in benign ones. Aafer et al. use these calls to form feature vectors and then apply various classifiers: ID5 DT, C4.5 DT, KNN, SVM. The best results have been shown by KNN classifier. The implemented tool DroidAPIMiner has shown false positive rate of 2.2% and approximately the same false negative rate.

Yerima et al. [13] used machine learning approaches based on Bayesian classification to solve the problem of Android malware detection. They applied Bayesian classification with 3 different feature sets composed of permission-based, code-based or mixed features also varying the number of selected features. They conclude that the mixed feature set gives the best results achieving the accuracy of 93.1%, false negative rate of 9.1% and false positive rate of 5.1%.

Similar approach was proposed by Sharma et al. [14] They applied Naive Bayesian and k-Nearest Neighbor classifiers on a feature set composed of permission-based and API call based features and also several techniques for feature selection to remove most of the redundant information. The authors assessed the results of classification according to the achieved accuracy and true positive rate (TPR) and revealed that kNN classifier achieved best accuracy while Naive Bayesian classifier reached the best TPR score. According to the results of their experiments, best false positive rate equal to 4.4% was achieved with kNN classifier using 15 features obtained via Information Gain Method.

Comparing the detection methods characteristics presented in the papers [9; 11; 12; 13; 14] the proposed method outperforms methods [13; 14] in achieved false positive rate. The methods [9; 11; 12] give better detection characteristics evaluated on their specific datasets. As for the explanation of classification results, the methods [9; 10; 11; 12] provide such an explanation however it is much less detailed and demonstrative than the one produced by our method. There is no information about getting clarification of the classification results in papers [13; 14].

[1] The implementation of the proposed method is publicly available at https://github.com/nastya/apk-analysis

[2] $F = 2 * \frac{precision * recall}{precision + recall}$

As all the above mentioned methods were trained and tested on different sets of malicious and benign applications, it is not the best practice to compare the resulting detection characteristics. At the same time, performing experiment-based comparison using one and the same dataset is a complicated task because both the source code and the complete dataset used in each work are usually not publicly available. Some authors provide their feature vectors built [9; 11] or the malware dataset [11] by request. Still this data is not sufficient because the benign dataset used for classification is also essential. Therefore in our paper we provide a detailed experiment-based comparison only with *adagio* tool [10] which is publicly available (see section VI).

III. Models construction

The application analysis proposed in our paper is basically a two-stage process: first we build models for the analyzed application using static analysis, and then we match the resulting models with the pre-built set of models which represent families of known malware. We use three types of models reflecting application behavior at a certain level of abstraction: the most general model type is *permission model*, Android Framework *API calls* model and *API calls chains* model represent the more precise application models.

A. Permission model

The Android operating system uses a permission system (application capabilities) to restrict application access to the software and hardware resources of the device [15]. All the permissions necessary for the proper functionality of the application should be requested in its manifest (file AndroidManifest.xml) in the apk-file.

Requested permissions might concern system resources as well as resources of 3rd party applications installed on a device. We analyzed permissions requested by malicious applications and built a list of 101 system permissions (permissions concerning framework resources). Our analysis is aimed at determining similarity with any of the known malicious samples therefore we did not extend this list of permissions with the permissions requested by the legitimate apps.

The permission model of an application is a vector $v \in \{0,1\}^{101}$ with each of the components corresponding to one of the permissions in a formed list (permissions are ordered).

$$v_p = \begin{cases} 1, \text{ if permission } p \text{ is requested} \\ 0 \text{ otherwise} \end{cases}$$

B. Model of Android Framework API calls

A more detailed notion of the application behaviour may be taken from the Android Framework API calls utilized by the app. To build this type of model we need to decompile the application, which is done with the help of the Androguard tool [16].

Au et al. conducted a study of the Android permission system and formed a mapping from permissions to the lists of API calls which require the corresponding permission to be executed [17]. We herein refer to such API calls as protected API calls. Considering the results of Au et al. we have analyzed API calls used by the apps in the malicious collection and have formed a list of the protected calls. As some malicious applications do not use any of the protected API calls, we extended the list with some unprotected API calls in such a manner that every malicious application uses at least 20 calls from the final list. [3] We have a total of 384 API calls in this list.

The model of Android Framework API call for an application is a vector $v \in \{0,1\}^{384}$ with each of the components corresponding to one of the API calls in the formed list (API calls are ordered).

$$v_f = \begin{cases} 1, \text{ if API call } f \text{ is used} \\ 0 \text{ otherwise} \end{cases}$$

C. Model of API call chains

The model of API call chains gives the most detailed description of application behavior of all the models. It is represented by a list of API call sequences (chains) where the calls are ordered by their expected order on application execution. The number of chains corresponds to the number of application *entry points* which are the application functions called by the framework on a certain event generated by the user or by the system. The simplest example of an entry point is the *onCreate* method of the application main Activity. Models of API calls chains are built using the decompiled application structure.

Model construction

a) Possible entry points discovery: We used the algorithm proposed by Lu et al. to find possible entry points [18]. It consists of the following steps:

1) The initial set of entry points consists of the application's main component (Activity, Service, Broadcast Receiver, Content Provider) methods called by the framework.
2) Using the current set of entry points, the call graph of the reachable function calls is built. [4]
3) All application methods overloading methods in Android Framework classes which are not reachable from the current entry points are added to the entry points set if the call graph built on the previous step contains the constructor of the corresponding class.
   Steps 2) and 3) are repeated until the entry points set remains the same after the iteration.

[3]The additional unprotected API calls are the calls that are rather common among both malicious and benign apps. These are the calls like: *Ljava/lang/StringBuilder;->append*, *Landroid/widget/TextView;->setText*, *Landroid/widget/Toast;->show* and others. There are many suitable variants of such a set of additional API calls. The complete list of the calls that we used can be seen in the implementation of our method.

[4]The function call graph is build based on the smali-code representation of the application considering *invoke** instructions of the methods. We used Androguard tool [16] to get the smali-code of each of the application methods and build on top of it our simple implementation of call graph construction. No complex analysis addressing the reflection calls or points-to analysis is performed in our current implementation.

b) Function call graph building for each of the entry points: Picking out the vertices corresponding to the entry points together with all the vertices and arcs reachable from them in a general graph of reachable calls we obtain function calls graphs for each of the entry points. For each vertex the outgoing arcs in graphs are ordered by the function call sequence in a method corresponding to a vertex.

c) API calls chains construction: Android Framework API calls are the leaf nodes in the built function call graphs. Apart from Android Framework API calls, API calls of known libraries compiled with the application are interesting for our analysis. The descendants of such vertices are excluded from the analysis as they might lead to commom chains in applications with totally different functionality using just the same library. Thus, for each function call graph in an application we form an API call chain by traversing it using a depth-first search algorithm and leaving in a chain Android Framework and known libraries API calls.

Figure 1 depicts function call graph of *onCreate* entry point of one of the application Services. The order of adjoining vertices traversal corresponds to the direction from left to right from top to bottom on this picture. The resulting API calls chain is:

Landroid/app/Service;->onCreate,
Landroid/os/PowerManager;->newWakeLock,
Landroid/os/PowerManager$WakeLock;->acquire,
Ljava/lang/Object;-><init>,
Landroid/database/sqlite/SQLiteOpenHelper;-><init>,
Landroid/os/Handler;->postDelayed

The models of API call chains for malicious applications may also contain legitimate chains which might be encountered in benign applications. However, long matches even in such chains are suspicious and are encountered quite rarely. Malware model chains containing its key functionality represent the main interest for analysis. Here is, for example, a subsequence of such chain in trojan application Mobiletx model. This application collects user data (IMSI number) and sends it via SMS-message to adversaries.

Landroid/content/Context;->getSystemService,
Landroid/telephony/TelephonyManager;->getSubscriberId,
Ljava/lang/StringBuilder;->append,
Ljava/lang/StringBuilder;->toString,
Landroid/telephony/SmsManager;->sendTextMessage

IV. Application analysis

The mobile application analysis proposed in this paper consists of model construction for the analyzed application and matching it with the pre-built malware models. Model construction is described in section III. Model matching is implemented in three steps, with each step corresponding to matching models of the same type. The matching process is described in detail further in this section. First, the permission model of the analyzed app is matched with the permission models of all malicious apps. This first step results in a list of malicious applications which have permission models similar to the analyzed one, which we will refer to as permission-similar apps. The models of the Android Framework API calls of the analyzed app and permission-similar malware apps are matched on the second step. This results in a list API-calls-similar apps. Finally, models of API call chains of the analyzed app and its API-calls-similar malicious apps are matched on the third step. The final result of our analysis is a list of malicious apps which turned out to be similar to the analyzed one regarding their API call chains. In this context the analyzed app is considered malicious if this resulting list contains at least one similar model.

The analysis results at each step are embedded in the results of the previous step due to models embeddedness. Indeed, the applications having similar API call chains certainly have similar sets of the used Android Framework API calls. And as the possibility of using most of the considered Android Framework API calls is determined by the required permissions, API-calls-similar apps are likely to be permission-similar. We should notice that for most apps the results of our three-step analysis are the same with the results of analysis implementing only the last step, matching analyzed model of API calls chains with all malware models of the same type. However, such analysis is much more time-consuming. Besides, the results obtained on the first two steps can also be a point of interest for malware analysts.

A. Permission model matching

Application permission model is represented by a binary vector with 101 components, whose component values correspond to a chosen set of permissions. Let us reference this set as *P*. We use weights for permissions, the weight of permission *p* is referenced as w_p. High weights are used for the most dangerous permissions such as *android.permission.SEND_SMS*, for the remaining permissions w_p is set to 1. Higher weights for dangerous permissions make the value of similarity function higher for analyzed apps requiring such permissions which does not necessarily correspond to a higher similarity between compared apps. However, this is indicating the app is suspicious and certainly needs further detailed analysis. [5]

To match permission model *v* of the analyzed app with permission model *u* of some malicious app, the similarity function is computed:

$$S_P(v,u) = \begin{cases} \frac{\sum\limits_{p\in P} w_p * 1_{v_p=u_p \& u_p=1}}{\sum\limits_{p\in P} 1_{u_p=1}}, \text{ if } \sum\limits_{p\in P} 1_{u_p=1} \neq 0 \\ 1 \text{ otherwise} \end{cases} \quad (1)$$

The applications are considered permission-similar if $S_P(v,u)$ exceeds a predefined $Threshold_{p_{sim}}$. According to

[5]In our implementation we used $w_p = 10$ for dangerous permissions, although the actual choice of high weights values can be varied widely. We did not explore comprehensively the dependency between similarity function values and the set of used weights. Considering that their only purpose is to make the resulting similarity value high enough, the weights for dangerous permission should just be set to some high values, e.g. ten fold higher than the usual weights or more ($w_p \geq 10$).

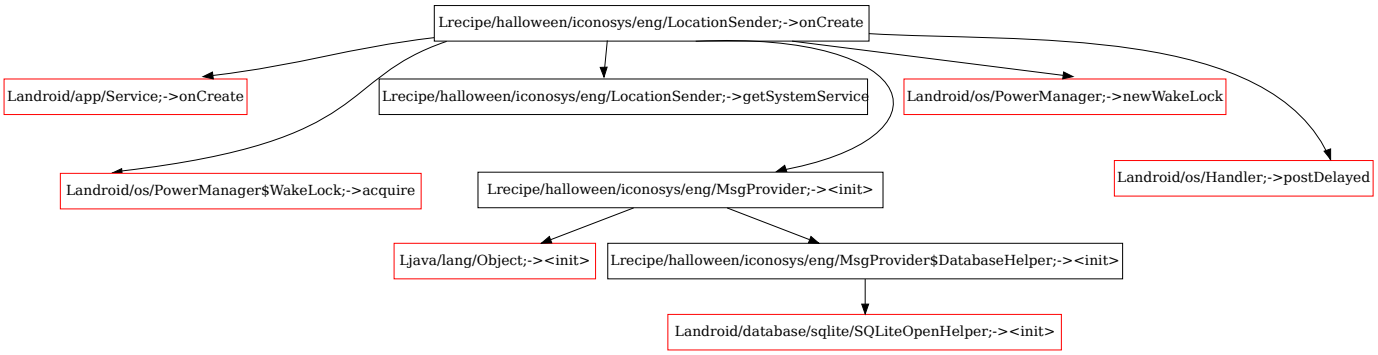

Figure 1. Function call graph. Vertices corresponding to API calls are coloured red.

(1) we consider all applications permission-similar with malicious apps not requiring any system permissions. Although it might seem strange for a malicious application not to require any permissions, the most common example of such a malware is an application utilizing root exploits.

B. Matching models of Android Framework API calls

Matching models of Android Framework API calls is performed the same way as permission model matching. Let us reference the chosen Android Framework API calls set as *A*. To match model of Android Framework API calls of the analyzed app with model of Android Framework API calls *u* of some malicious app, the similarity function is computed:

$$S_{API}(v,u) = \frac{\sum\limits_{f \in A} 1_{v_f = u_f \& u_f = 1}}{\sum\limits_{f \in A} 1_{u_f = 1}} \qquad (2)$$

The applications are considered API-calls-similar if $S_{API}(v,u)$ exceeds a predefined $Threshold_{API_{sim}}$. [6]

C. Matching models of API calls chains

Matching models of API call chains is based on finding the longest common subsequence of chain pairs. Let us denote a set of chains in application *u* model as $chains(u)$. For each pair of chains c_v and c_u we compute the value $lcs(c_v, c_u)$ which is equal to the number of elements in their longest common subsequence. The longest common subsequence of chains c_v and c_u itself we denote as $cs(c_v, c_u)$. The chains c_v and c_u are considered to have a *common subchain* if their longest common subsequence is long enough, i.e. the relative value $\frac{lcs(c_v,c_u)}{|c_u|}$ exceeds $Threshold_{common}$ and the absolute value $lcs(c_v, c_u)$ is at least $Threshold_{common_{abs}}$ API-calls.

The chains of the analyzed application are processed in order of decreasing length. For each of the chains c_v in the analyzed app model that have *common sub-chains* with chains in malware model we choose the chain $c_{u_{similar}}$ which gives the longest *common subchain*, if it is not unique we consider all such chains. The common subchain is fragmented by chain c_v on *components* — continuous areas. [7] Figure 2 depicts chains as sequences of symbols, the top one corresponds to one of the chains in the analyzed app, and the bottom one corresponds to one of the chains in malicious app. We colored their common subchain with the minimal number of components, in this case it is fragmented on 2 components. We denote a number of components on which the common subchain of chains c_u and c_v is fragmented as $components(c_v, c_u)$. The less *components* in common subchain fragmentation, the higher similarity between c_v and c_u. At the same time, if we set too rigorous a restriction on a number of components, it can lead to the analyzer ignoring similar chains having just minimal insignificant differences (such as adding API calls not connected with the main functionality of the chain). Therefore, we considered only such chains $c_{u_{match}}$ from the set $c_{u_{similar}}$ which satisfy the following inequality:

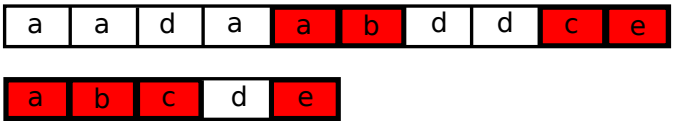

Figure 2. Fragmentation of a chain on components.

$$components(c_v, c_{u_{match}}) < \left\lceil \frac{1}{3} lcs(c_v, c_{u_{match}}) \right\rceil \qquad (3)$$

To satisfy this inequality a common subchain needs to have a continuous area of at least three elements (3 is a reciprocal of the coefficient on the right-hand side of the inequality). If the inequality (3) is satisfied by several chains in the malware model, we choose the one containing protected API calls (if there are no such chains, we choose any chain). Thus, for each chain c_v we can choose no more than one chain in the malware model. Correspondingly, chains $cs(c_v, c_{u_{match}})$ form a set of *common chains* for compared models. The result of model matching is represented by 4 values:

[6] It is also possible to introduce weights for API calls and to use higher weights for the dangerous ones. However, we did not explore this topic, because the results of comparison obtained using the equation (2) were satisfactory for this intermediate step of our analysis.

[7] If there are several common subchains (corresponding to several different longest common subsequences) we choose the one fragmented on a minimal number of components.

1) a – the number of common chains;
2) b – the total length of common chains;
3) c – the number of *long common chains*;
   The chain is considered *long* if its number of elements is not less than a threshold set in our analysis to $Threshold_{long_{chain}}$.
4) d – the number of common chains containing protected API calls.

The more each of these values, the higher similarity of the compared models respectively. We consider applications API-chains-similar if at least one of the following conditions is held:

- $a \geq Threshold_{amount}$ and $b \geq Threshold_{length}$
  This condition means the applications have several common subchains with the considerable total length;
- $c \geq 2$ — the applications have at least two long common subchains;
- $c \geq 1$ and $d \geq 1$ — the applications have at least one long common subchain and at least one common subchain with protected API-calls;
- $d \geq 1$ and $b \geq Threshold_{length}$
  This condition means the applications have a considerable total length of common subchains and at least one of them contains protected API-calls;
- $\frac{a}{|chains(u)|} \geq 0.95$ and $\frac{b}{\sum\limits_{c \in chains(u)} length(c)} \geq 0.95$.
  We use the last condition to detect similarity with models which have a small number of API call chains or a short total length of chains. Additionally we check if the chains in the compared models are completely equal independently of their length. The apps with equal chains are considered API-chains-similar. This is done to support reflexivity feature of model relation, i.e. every app is API-chains-similar with itself.

In the conditions listed above $|chains(u)|$ is a number of API call chains in application model u; $length(c)$ is the length (number of API calls) of chain c.

D. Thresholds tuning

All the threshold values and the coefficient on the right side of inequality (3) were determined by a series of experiments on a part of malicious and benign applications collections described in experiments section. Table I gives the obtained thresholds that we used further in our experiments. Those values can be tuned by the users of our tool themselves, however changing them might lead to detection coverage or false positives rate changing.

As our method depends on a large number of threshold values, it is almost impossible to explore the influence of all the threshold values at once because we need to carry out too many experiments to get this multivariable function and finally find her maxima giving us the best false positive/false negative ratio. Therefore we studied this dependency by changing each of the threshold values while having all the others fixed and choosing the value giving the best results in the context of false negatives/false positives ratio.

Table I
USED THRESHOLDS

Threshold	Value
$Threshold_{p_{sim}}$	0.9
$Threshold_{API_{sim}}$	0.7
$Threshold_{common}$	0.85
$Threshold_{common_{abs}}$	3
$Threshold_{long_{chain}}$	30
$Threshold_{amount}$	7
$Threshold_{length}$	50

The thresholds $Threshold_{p_{sim}}$ and $Threshold_{API_{sim}}$ are aimed to keep false negative rate close to zero while minimizing the number of permission-similar and API-similar models to be used in comparison on the last and the most time-consuming step of analysis. For tuning them we used half of the *Drebin* collection to generate models and 300 more applications from this collection to assess the results of our method (all these application were chosen randomly). While tuning $Threshold_{p_{sim}}$ we omitted second step of analysis directly comparing API-call chains models for found permission-similar apps. We found out that the number of false negatives ratio remains the same for $Threshold_{p_{sim}} \in [0.1, 0.9]$ and it is equal to 4.6% and when we set $Threshold_{p_{sim}}$ to 0.95 the false negatives ratio slightly grows to 5%. At the same time the average number of permission-similar models drops from 1929 ($Threshold_{p_{sim}}$ = 0.1) to 412 ($Threshold_{p_{sim}}$ = 0.9). Therefore we have set $Threshold_{p_{sim}}$ to 0.9.

While tuning $Threshold_{API_{sim}}$ we used the same logic: keeping false negatives rate at the lowest possible level while decreasing the average number of API-similar models. On the plot depicted on figure 3 we can see the best point with 4.6% false negative ratio and 142 API-similar models on average. This point corresponds to $Threshold_{API_{sim}}$ set to 0.7.

The thresholds for comparing API-call chains were obtained manually by comparing a number of malicious applications from the same malware families. The obtained values were further checked on a part of benign collection to ensure they do not produce a large number of false positives.

V. EXPERIMENTS

In this section we describe experiments carried out on our implemented method. First we applied our method to binary classification and measured the false positive and false negative rate. Second, we used the proposed method to analyze the malware collections and prove the ability of the method to detect similarities among apps from the same malware family. At the same time we used this analysis to form the smallest possible set of malware models covering all malware collections we used. We also measured false positive rate of the method on a large collection of benign applications.

A. Binary classification

We used two malware datasets for our experiments: *Drebin* collection provided by Arp et al. [11]; and *ISCX* collection provided by ISCX, University of New Brunswick [19], more

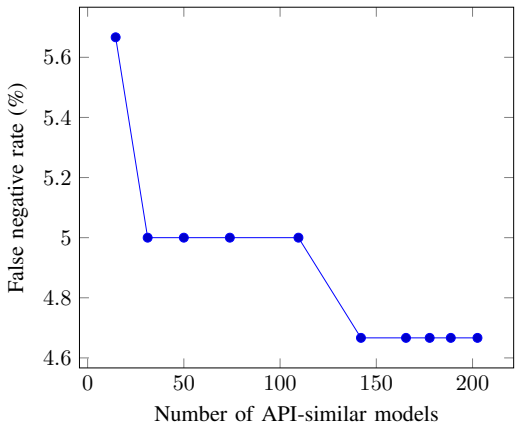

Figure 3. Dependence of the false negative rate and an average number of API-similar models on $Threshold_{API_{sim}}$.

specifically, we considered only its Android botnet part. The total number of apps in both collections is 7489 however our method failed to build models for 40 applications and some of the built models are not functional because they don't contain any API-calls chains. We discuss it in detail in the further subsection. Therefore we used a malware dataset of 7449 applications in total. As for benign applications we used 3000 of applications downloaded from Google Play, our method failed to build models for 7 of those applications. We have chosen 2/3 of the malicious applications randomly to form malware models and tested our method on the rest of the malware collection and all benign applications. We conducted this experiment thrice each time choosing malware models randomly.

Table II
RESULTS OF APPLICATION ANALYSIS (BINARY CLASSIFICATION)

Number of models used	Alarms on malware collection	Alarms on benign collection	FN	FP
4787	2254 / 2483	63 / 2993	9.3%	2.1%
4787	2240 / 2483	51 / 2993	9.7%	1.7%
4787	2262 / 2483	46 / 2993	9.1%	1.5%

In this experiment our method shows the false negative rate of less than 10% and the false positive rate of less than 2% on average (table II). We do not provide the results of intermediate steps of analysis here as they are not essential for assessing method detection coverage and false alarms.

B. Analysis of malware collections

We used our method to explore the structure of two above-mentioned datasets: Drebin and ISCX. By exploration here we mean finding similarities and dependencies between apps in collections. Our main goal was to get a set of applications from each of the collections such that if taken as a set of malware models it would cover the rest of the collection, i.e. each app from a malware dataset would be considered chain-similar with one of the malware models.

1) Drebin collection: The collection consists of 5560 applications belonging to 179 malware families. All samples were collected in the period of August 2010 to October 2012. Androguard tool failed to decode 10 of them due to some bugs within the tool or improper apk-files (6 samples were not recognized as zip archive data by *file* utility). For 28 of the rest 5550 apps our method failed to build any API call chains. We manually analyzed those apps dynamically and revealed that 25 of them do not run in the Android emulator under Droidbox analysis [20]. We also analyzed most of them statically and encountered malformed smali-code: it either did not contain any instructions of the methods or the names used in the manifest did not match the actual class names of the app. For 3 apps our method failed to build chains because their package names match the names of some known libraries and we do not consider such code while building models. This limitation of our method is described in detail in sections VII and V-D. Thus, we excluded these 38 apps from analysis.

Based on found chain-similarities we can take 422 apps such that they cover 5131 apps from the dataset (including those 422). For 381 apps our method did not find any chain-similar apps in the dataset. We further analyzed clusters of similar apps formed by these 422 apps. We matched clusters with the preliminary provided malware families marks such that the resulting mark of the cluster was mark of the majority of apps forming it. We then computed a number of apps which mark did not match with the mark of the cluster: 3.1% (159 out of 5131) of the apps were clustered incorrectly. Therefore, to form set of apps to be used as malware models we have taken 422 + (5560 - 38 - 5131) = 813 applications from the Drebin collection.

Apart from this clustering analysis we also performed basic analysis of the dataset aiming to get a notion of programming practices and obfuscation methods used by apps in the dataset. The results reveal that 37% of apps in collection contain embedded advertisements; the apps are mostly of a small size, 50% of the apps contain less than 50 classes. Analysis of the longest API calls chain per app revealed that for 50% of the apps its length is less than 70, whereas for 92% of the apps its length is less than 200. 60.5% of the apps use reflection; 38% use API related to native code execution; 32% invoke cryptographic API (we considered only javax.crypto package). Although our method is not adapted to native code or reflection usage, it is still able to find similarity between such apps leaning on reflection/native code execution API calls (*Ljava/lang/Class;->forName, Ljava/lang/reflect/Method;->invoke, Ljava/lang/Runtime;->exec ...*) encountered in chains and other parts of the app which do not use this techniques. We also analyzed 200 randomly chosen apps manually to get a notion of names obfuscation usage in the dataset: 44% did not use any name obfuscation at all; part of class names in 51.5% of the apps was obfuscated, most of them were in libraries compiled into app; and only 4.5% of the apps used obfuscation of all names. In general our manual analysis has revealed that most apps in the dataset can be more or less easily reverse-engineered

and do not use heavy obfuscation.

2) ISCX collection: The ISCX collection was analyzed analogously to the Drebin one. This collection consists of 1929 samples belonging to 14 malware families. The collection includes more recent malware: the samples were collected from 2010 till 2014. Androguard failed to decompile 30 apps from the dataset but this time the problem was solely on the Androguard side. Our method failed to build chains for 66 samples where 19 apps consisted of packages matching with known library names. Analysis of the rest of the apps generating no API calls chains revealed one more case of malformation: in some apps all method names (including methods like *onCreate*, *onStart*) were modified with adding suffix 'abc123' which caused it not to launch any components properly. Thus, we excluded 96 apps from the further analysis.

For the rest of the collection our method formed 120 clusters (with the length of at least 2 apps) covering 1723 apps of the collection. Other apps in the collection were not clustered and for 114 apps our method did not find any other API-chain-similar samples in the collection. 3.9% (68 out of 1723) of the apps were clustered incorrectly, i.e. their malware family mark did not match the mark of majority of apps in a cluster. Thus, we can form a set of malware models using 120 + (1929-96-1723) = 230 applications from the collection.

Basic analysis of ISCX collection revealed that the apps in the dataset are generally more complex and large than the apps from Drebin collection: only 30% of apps contain less that 50 classes, almost 50% contain more than 100 classes. 18% of the apps embed advertisements, 76% use reflection API, 62% use native code execution and more than 53% use cryptographic API. We analyzed 100 randomly chosen samples from the collection manually and revealed that 29% do not use any name obfuscation, 70% of apps obfuscate part of class names belonging mostly to advertisement or other libraries and only in 1 analyzed app all names were obfuscated.

C. Large-scale benign application analysis

The proposed mobile application analysis method was tested on a collection of 43819 mobile apps downloaded from Google Play between 2013 and 2016 using 850 apps from the Drebin and ISCX collections as malware models. The apps used as malware models were obtained from a union of models for each collection excluding duplicate and similar apps – the collections partly overlapped. In this experiment we have set $Threshold_{p_{sim}}$ to 0.7 instead of 0.9 to reveal more possible false positives caused by API-chains-similarity. Table III provides the analysis results.

Table III
RESULTS OF BENIGN APPLICATIONS ANALYSIS

Step of analysis	Considered as malicious	FP
Permissions (1)	43357 / 43819	98.9%
Android Framework API calls (2)	39451 / 43819	90.0%
API calls chains (3)	1388 / 43819	3.2%

Thus, the implemented method false positive rate (type II error) is 3.2%. This number is a little bit higher than the one obtained in our experiment with binary classification because we used another value for $Threshold_{p_{sim}}$. Still this value better reflects the resilience of our method to generating false alarms because the main part of the analysis is matching API calls chains. The results of intermediate steps (Permissions (1) and Android Framework API calls (2)) show the amount of analyzed apps which have at least one similar app matching it with the corresponding type of malware models. As can be seen from these intermediate results a set of used permissions and API calls does not determine the application behavior precisely enough to be used for malware detection. The first and the second steps of analysis are mainly used for reduction of malware models number to be matched with on the last, the most time-consuming step.

D. Analysis of misclassified applications

We manually reviewed part of benign applications from Google Play, which caused our analyzer to generate false alarms. Most false positives were caused by common code in library packages whose names were obfuscated and therefore they were not taken by our analyzer as library code. As we have mentioned in section III we consider API calls to known libraries in building API-call chain models but exclude the inner code of libraries from analysis. To accomplish this goal, we formed a list of known library packages and filtered them in our experiments by names. We also implemented filtering library packages by their method hashes. However, we have not managed to form a proper filter comprising all necessary hashes as this is a rather challenging task. Nevertheless the users of our method have the opportunity to utilise their own bloom filter with library method hashes and thus get rid of this drawback of our method.

Some benign applications had large parts of the common code with malware models, in this case, it is most likely the corresponding malicious application was created based on the earlier version of the benign application and was repackaged with additional functionality. Another part of applications contained methods similar with some methods in malicious applications not connected with its malicious activity (for example, serialization and deserialization of XML-models).

VI. COMPARISON WITH *adagio*

Adagio is a static Android application analysis method proposed by H. Gascon et al [10]. We have chosen this method for our comparison because of the open source code of *adagio* [21] and its similarity with our method in its basics.

Adagio implements machine learning algorithm utilizing static feature set. The feature vectors are based on function call graphs of the applications similarly with our static models. However, Gascon et al. represent function call graphs differently, they consider all instructions of methods in function call graph, and the dependencies between them to form a set of hashes later embedded in feature space. And in our method we consider Android Framework and known library API-calls

corresponding to parameters in only *invoke** instructions of the methods.

We conducted the same binary classification experiment thrice on a collection of 7449 malicious applications from Drebin and ISCX collections and 3000 benign applications splitting 2/3 of it as a training dataset and 1/3 as a testing dataset. Unfortunately, the method failed to build feature vectors for 509 malicious and 11 benign applications. Table IV provides the results of this experiment: the number of malware and benign models varies slightly between experiments because 2/3 of the models for training dataset were chosen randomly from the union of malicious and benign feature vectors.

Table IV
RESULTS OF APPLICATION ANALYSIS FOR *adagio*

Malware models	Benign models	FN	FP
4572	1983	45 / 2368 (1.9%)	93 / 1006 (9.2%)
4571	1984	51 / 2369 (2.2%)	97 / 1005 (9.7%)
4579	1976	57 / 2361 (2.4%)	90 / 1013 (8.9%)

Adagio tool using default configuration values shows the false negative rate of 2.2% and false positive rate of 9.3% on average in this experiment. Therefore our method has a lower detection rate with the randomly chosen models but at the same time it is significantly more reliable considering generated false alarms.

Considering the explanation of classification, for each supposedly malicious application *adagio* lists the methods in the call graph with the highest weights in the decision function. Thus all the explanation is a list of methods in the app that the tool considers suspicious (possibly malicious) without explaining what is indeed suspicious in those methods. Our method oppositely gives rather detailed information represented by all matched API call chains in the analyzed application and the concrete malware model.

As for the runtime performance of compared methods, *adagio* is able to analyse applications at a much higher speed than our method. The analysis time is about 1 second for adagio whereas for our method it is about 2.3 seconds in case of malicious application and about 23.8 seconds in case of benign application taken as input. At the same time our method takes less time for generating models: 1.5 seconds vs 4.4 seconds for adagio in case of a malicious application and 11.2 vs 20.5 seconds in case of a benign application [8].

VII. LIMITATIONS AND FUTURE WORK

Concluding all the above-mentioned observations, we believe our method provides rather in-depth static analysis giving detailed explanation of its results. However, it requires a proper choice of malware models to keep high detection rate and appropriate runtime performance as the method detection coverage significantly depends on the malicious applications used for model mining. It is worth noticing that this is a general problem for almost all malware detection methods, they need constant retraining/updating models set. At the same time, the malware models set should be as compact as possible as the time required for our analysis grows linearly with the number of malware models.

The best way to improve the method performance considering false positives is to filter library packages statically compiled in applications properly. This can be done by method hash-based filtering, however, the filter should be trained on a large number of benign applications to comprise hashes of all known library methods (besides, all versions of each library should be represented).

The method can probably be enhanced by changing the algorithm of API call chains construction or the algorithm of their matching. For example, in our proposed method vertex of function call graphs corresponding to application classes methods are unique, and therefore such a vertex corresponds to a single part of the resulting API call chain even when it is called from several other methods. Replication of such parts in chains for each of the call points would essentially increase their length (and the time of analysis) but it can possibly increase the method precision. The proposed method does not consider application component interaction via sending Intents. Considering them would add connections between API call chains in corresponding models and it might also increase the method precision. The method can be complemented with dynamic analysis considering the actually observed API call chains in order to detect obfuscated malicious samples.

VIII. CONCLUSION

In this paper we proposed a method for analysis and classification of Android applications. The method is based on system permissions and API calls used by the application. Experiments carried out on collections of malicious applications show that the method detects similar apps for 90–94% of apps in the datasets. The testing method on a collection of applications from Google Play shows a false positive rate of about 3.2%. The method may be used by mobile application marketplaces as a part of a complex analysis or by malware analysts as a helper tool to reduce the time required for manual analysis. Regarding the obfuscation, our results show that explored Android malware rarely uses obfuscation or encryption techniques to make static analysis more difficult, which is quite the opposite of what we see in the case of the 'Wintel' endpoint platform family and we should expect the situation to change in the near future.

We also present experiment-based comparison with the state-of-the-art Android malware detection method *adagio*. The proposed method outperforms *adagio* in revealed true positive rate but has smaller detection coverage with randomly chosen malware models.

[8]The runtime performance was measured on a desktop PC with Intel(R) Xeon(R) CPU E31225 @ 3.10GHz processor utilizing only one of its processing units

ACKNOWLEDGMENT

We would like to thank our shepherd Douglas Stebila for his help in improving our paper and Erinn Clark for the language editing and useful remarks. We also would like to thank Daniel Arp and Natalia Stakhanova for providing access to Drebin and ISCX Android malware collections.

REFERENCES

[1] A. Skovoroda and D. Gamayunov, "Securing mobile devices: malware mitigation methods," *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, vol. 6, no. 2, pp. 78–97, 2015.

[2] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "PiOS: Detecting Privacy Leaks in iOS Applications," in *Proc. of the 18th Annual Network and Distributed System Security Symposium (NDSS 2011)*. The Internet Society, 2011.

[3] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *Proc. of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI 2010)*. USENIX Association, 2010, pp. 1–6.

[4] A. Shabtai, L. Tenenboim-Chekina, D. Mimran, L. Rokach, B. Shapira, and Y. Elovici, "Mobile malware detection through analysis of deviations in application network behavior," *Computers & Security*, vol. 43, pp. 1–18, 2014.

[5] H. Kim, J. Smith, and K. G. Shin, "Detecting Energy-greedy Anomalies and Mobile Malware Variants," in *Proc. of the 6th International Conference on Mobile Systems, Applications, and Services (MobiSys 2008)*. ACM, 2008, pp. 239–252.

[6] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "RiskRanker: Scalable and Accurate Zero-day Android Malware Detection," in *Proc. of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys 2012)*. ACM, 2012, pp. 281–294.

[7] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based Detection of Android Malware Through Static Analysis," in *Proc. of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, 2014, pp. 576–587.

[8] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The Evolution of Android Malware and Android Analysis Techniques," *ACM Computing Surveys*, vol. 49, no. 4, pp. 76:1–76:41, 2017.

[9] E. Mariconti, L. Onwuzurike, P. Andriotis, E. D. Cristofaro, G. J. Ross, and G. Stringhini, "MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models," in *Proc. of the 24th Annual Network and Distributed System Security Symposium (NDSS 2017)*. The Internet Society, 2017.

[10] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural Detection of Android Malware Using Embedded Call Graphs," in *Proc. of the 2013 ACM Workshop on Artificial Intelligence and Security (AISec 2013)*. ACM, 2013, pp. 45–54.

[11] D. Arp, M. Spreitzenbarth, M. Huebner, H. Gascon, and K. Rieck, " Drebin: Efficient and Explainable Detection of Android Malware in Your Pocket," in *Proc. of the 21st Annual Network and Distributed System Security Symposium (NDSS 2014)*. The Internet Society, 2014.

[12] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android," in *Security and Privacy in Communication Networks*, T. Zia, A. Zomaya, V. Varadharajan, and M. Mao, Eds. Springer International Publishing, 2013, pp. 86–103.

[13] S. Y. Yerima, S. Sezer, and G. McWilliams, "Analysis of Bayesian Classification based Approaches for Android Malware Detection," *IET Information Security*, vol. 8, no. 1, pp. 25–36, 2014.

[14] A. Sharma and S. K. Dash, "Mining API Calls and Permissions for Android Malware Detection," in *Proc. of the 13th International Conference on Cryptology and Network Security*, D. Gritzalis, A. Kiayias, and I. Askoxylakis, Eds., vol. 8813. Springer International Publishing, 2014, pp. 191–205.

[15] Google Inc. official documentation, "System permissions," http://developer.android.com/intl/ru/guide/topics/security/permissions.html, access date: 26.07.2017.

[16] A. Desnos et al, "Androguard," https://github.com/androguard/androguard, access date: 26.07.2017.

[17] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android Permission Specification," in *Proc. of the 2012 ACM Conference on Computer and Communications Security (CCS 2012)*. ACM, 2012, pp. 217–228.

[18] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities," in *Proc. of the 2012 ACM Conference on Computer and Communications Security (CCS 2012)*. ACM, 2012, pp. 229–240.

[19] A. F. A. Kadir, N. Stakhanova, and A. A. Ghorbani, "Android Botnets: What URLs are Telling Us," in *Proc. of Network and System Security - 9th International Conference (NSS 2015)*, M. Qiu, S. Xu, M. Yung, and H. Zhang, Eds. Springer International Publishing, 2015, pp. 78–91.

[20] P. Lantz, "DroidBox: Android Application Sandbox," https://github.com/pjlantz/droidbox, access date: 26.07.2017.

[21] H. Gascon, "Adagio," https://github.com/hgascon/adagio, access date: 26.07.2017.