MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models

Enrico Mariconti[†], Lucky Onwuzurike[†], Panagiotis Andriotis[‡],
Emiliano De Cristofaro[†], Gordon Ross[†], and Gianluca Stringhini[†]
[†]University College London, [‡]University of the West of England
{enrico.mariconti.14,lucky.onwuzurike.13,e.decristofaro,g.ross,g.stringhini}@ucl.ac.uk,
panagiotis.andriotis@uwe.ac.uk

***Abstract*—The rise in popularity of the Android platform has resulted in an explosion of malware threats targeting it. As both Android malware and the operating system itself constantly evolve, it is very challenging to design robust malware mitigation techniques that can operate for long periods of time without the need for modifications or costly re-training. In this paper, we present MaMaDroid, an Android malware detection system that relies on app behavior. MaMaDroid builds a behavioral model, in the form of a Markov chain, from the sequence of abstracted API calls performed by an app, and uses it to extract features and perform classification. By abstracting calls to their packages or families, MaMaDroid maintains resilience to API changes and keeps the feature set size manageable. We evaluate its accuracy on a dataset of 8.5K benign and 35.5K malicious apps collected over a period of six years, showing that it not only effectively detects malware (with up to 99% F-measure), but also that the model built by the system keeps its detection capabilities for long periods of time (on average, 86% and 75% F-measure, respectively, one and two years after training). Finally, we compare against DroidAPIMiner, a state-of-the-art system that relies on the frequency of API calls performed by apps, showing that MaMaDroid significantly outperforms it.**

I. Introduction

In the first quarter of 2016, 85% of smartphone sales were devices running Android [49]. Due to its popularity, cyber-criminals have increasingly targeted this ecosystem [17], as malware running on mobile devices can be particularly lucrative – e.g., allowing attackers to defeat two factor authentication [51], [53] or trigger leakage of sensitive information [27]. Detecting malware on mobile devices presents additional challenges compared to desktop/laptop computers: smartphones have limited battery life, making it infeasible to use traditional approaches requiring constant scanning and complex computation [43]. Therefore, Android malware detection is typically performed by Google in a centralized fashion, i.e., by analyzing apps submitted to the Play Store using a tool called Bouncer [40]. However, many malicious apps manage to avoid detection [1], and anyway Android's openness enables manufacturers and users to install apps that come from third-party market places, which might not perform any malware checks at all, or anyway not as accurately [67].

As a result, the research community has devoted significant attention to malware detection on Android. Previous work has often relied on the permissions requested by apps [20], [46], using models built from malware samples. This strategy, however, is prone to false positives, since there are often legitimate reasons for benign apps to request permissions classified as dangerous [20]. Another approach, used by DroidAPIMiner [2], is to perform classification based on API calls frequently used by malware. However, relying on the most common calls observed during training prompts the need for constant retraining, due to the evolution of malware and the Android API alike. For instance, "old" calls are often deprecated with new API releases, so malware developers may switch to different calls to perform similar actions, which affects DroidAPIMiner's effectiveness due to its use of specific calls.

In this paper, we present a novel malware detection system for Android that instead relies on the *sequence* of *abstracted* API calls performed by an app rather than their use or frequency, aiming to capture the behavioral model of the app. Our system, which we call MaMaDroid, abstracts API calls to either the *package* name of the call (e.g., `java.lang`) or its source (e.g., `java`, `android`, `google`), which we refer to as *family*. Abstraction provides resilience to API changes in the Android framework as families and packages are added and removed less frequently than single API calls. At the same time, this does not abstract away the behavior of an app: for instance, packages include classes and interfaces used to perform similar operations on similar objects, so we can model the types of operations from the package name, independently of the underlying classes and interfaces. For example, we know that the `java.io` package is used for system I/O and access to the file system, even though there are different classes and interfaces provided by the package for such operations.

After abstracting the calls, MaMaDroid analyzes the *sequence* of API calls performed by an app, aiming to model the app's behavior. Our intuition is that malware may use calls for different operations, and in a different order, than benign apps. For example, android.media.MediaRecorder can be used by any app that has permission to record audio, but the call sequence may reveal that malware only uses calls from this class *after* calls to getRunningTasks(), which allows recording conversations [65], as opposed to benign apps where calls from

NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA
Copyright 2017 Internet Society, ISBN 1-891562-46-0
http://dx.doi.org/10.14722/ndss.2017.23353

the class may appear in *any* order. Relying on the sequence of abstracted calls allows us to model behavior in a more complex way than previous work, which only looked at the presence or absence of certain API calls or permissions [2], [5], while still keeping the problem tractable [33]. MAMADROID builds a statistical model to represent the transitions between the API calls performed by an app, specifically, we model these transitions as Markov chains, and use them to extract features and perform classification (i.e., labeling apps as benign or malicious). Calls are abstracted to either their package or their family, i.e., MAMADROID operates in one of two modes, depending on the abstraction granularity.

We present a detailed evaluation of both classification accuracy (using F-measure, precision, and recall) and runtime performance of MAMADROID, using a dataset of almost 44K apps (8.5K benign and 35.5K malware samples). We include a mix of older and newer apps, from October 2010 to May 2016, verifying that our model is robust to changes in Android malware samples and APIs. To the best of our knowledge, this is the largest malware dataset used to evaluate an Android malware detection system in a research paper. Our experimental analysis shows that MAMADROID can effectively model both benign and malicious Android apps, and perform an efficient classification on them. Compared to other systems such as DROIDAPIMINER [2], our approach allows us to account for changes in the Android API, without the need to frequently retrain the classifier.

We show that MAMADROID is able to effectively detect unknown malware samples not only in the "present," (with F-measure up to 99%) but also consistently over the years (i.e., when the system is trained on older samples and classification performed over newer ones), as it keeps an average detection accuracy, evaluated in terms of F-measure, of 86% after one year and 75% after two years (as opposed to 46% and 42% achieved by DROIDAPIMINER [2]). We also highlight that when the system is not efficient anymore (when the test set is newer than the training set by more than two years), it is as a result of MAMADROID having low recall, but maintaining high precision. We also do the opposite, i.e., training on newer samples and verifying that the system can still detect old malware. This is particularly important as it shows that MAMADROID can detect newer threats, while still identifying malware samples that have been in the wild for some time.

Summary of Contributions. First, we introduce a novel approach, implemented in a tool called MAMADROID, to detect Android malware by abstracting API calls to their package and family, and using Markov chains to model the behavior of the apps through the sequences of API calls. Second, we can detect unknown samples on the same year of training with an F-measure of 99%, but also years after training the system, meaning that MAMADROID does not need continuous re-training. Our system is scalable as we model every single app independently from the others and can easily append app features in a new training set. Finally, compared to previous work [2], MAMADROID achieves significantly higher accuracy with reasonably fast running times, while also being more robust to evolution in malware development and changes in the Android API.

Paper Organization. The rest of the paper is organized as follows. The next section presents the MAMADROID system, then, Section III introduces the datasets used in our evaluation (Section IV), while Section V further discusses our results as well as its limitations. After reviewing related work in Section VI, the paper concludes in Section VII.

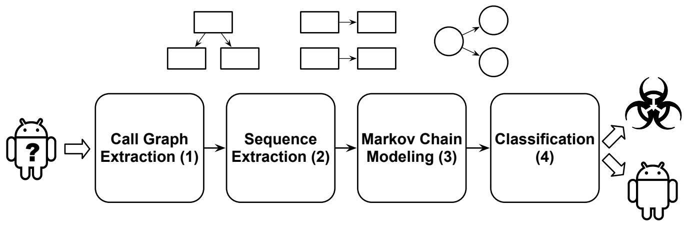


Fig. 1: Overview of MAMADROID operation. In (1), it extracts the call graph from an Android app, next, it builds the sequences of (abstracted) API calls from the call graph (2). In (3), the sequences of calls are used to build a Markov chain and a feature vector for that app. Finally, classification is performed in (4), labeling the app as benign or malicious.

II. THE MAMADROID SYSTEM

A. Overview

We now introduce MAMADROID, a novel system for Android malware detection. MAMADROID characterizes the transitions between different API calls performed by Android apps – i.e., the sequence of API calls. It then models these transitions as Markov chains, which are in turn used to extract features for machine learning algorithms to classify apps as benign or malicious. MAMADROID does not actually use the sequence of *raw* API calls, but abstracts each call to either its package or its family. For instance, the API call getMessage() is parsed as:

$$\underbrace{\overbrace{\underbrace{\text{java}}_{\textbf{family}}\text{.lang}}^{\textbf{package}}\text{.Throwable: String getMessage()}}_{\textbf{API call}}$$

Given these two different types of abstractions, we have two modes of operation for MAMADROID, each using one of the types of abstraction. We test both, highlighting their advantages and disadvantages — in a nutshell, the abstraction to family is more lightweight, while that to package is more fine-grained.

MAMADROID's operation goes through four phases, as depicted in Fig. 1. First, we extract the call graph from each app by using static analysis (1), next we obtain the sequences of API calls for the app using all unique nodes in the call graph and associating, to each node, all its child nodes (2). As mentioned, we abstract a call to either its package or family. Finally, by building on the sequences, MAMADROID constructs a Markov chain model (3), with the transition probabilities used as the feature vector to classify the app as either benign or malware using a machine learning classifier (4). In the rest of this section, we discuss each of these steps in detail.

B. Call Graph Extraction

The first step in MAMADROID is to extract the app's call graph. We do so by performing static analysis on the app's

```
package com.fa.c;

import android.content.Context;
import android.os.Environment;
import android.util.Log;
import com.stericson.RootShell.execution.Command;
import com.stericson.RootShell.execution.Shell;
import com.stericson.RootTools.RootTools;
import java.io.File;

public class RootCommandExecutor {
  public static boolean Execute(Context paramContext) {
    paramContext = new Command(0, new String[] { "cat " + Environment.
       getExternalStorageDirectory().getAbsolutePath() + File.separator + Utilities
       .GetWatchDogName(paramContext) + " > /data/" + Utilities.GetWatchDogName(
       paramContext), "cat " + Environment.getExternalStorageDirectory().
       getAbsolutePath() + File.separator + Utilities.GetExecName(paramContext) + "
        > /data/" + Utilities.GetExecName(paramContext), "rm " + Environment.
       getExternalStorageDirectory().getAbsolutePath() + File.separator + Utilities
       .GetWatchDogName(paramContext), "rm " + Environment.
       getExternalStorageDirectory().getAbsolutePath() + File.separator + Utilities
       .GetExecName(paramContext), "chmod 777 /data/" + Utilities.GetWatchDogName(
       paramContext), "chmod 777 /data/" + Utilities.GetExecName(paramContext), "/
       data/" + Utilities.GetWatchDogName(paramContext) + " " + Utilities.
       GetDeviceInfoCommandLineArgs(paramContext) + " /data/" + Utilities.
       GetExecName(paramContext) + " " + Environment.getExternalStorageDirectory().
       getAbsolutePath() + File.separator + Utilities.GetExchangeFileName(
       paramContext) + " " + Environment.getExternalStorageDirectory().
       getAbsolutePath() + File.separator + " " + Utilities.GetPhoneNumber(
       paramContext) });
    try {
      RootTools.getShell(true).add(paramContext);
      return true;
    }
    catch (Exception paramContext) {
     Log.d("CPS", paramContext.getMessage());
    }
    return false;
  }
}
```

Fig. 2: Code snippet from a malicious app (com.g.o.speed.memboost) executing commands as root.

apk.[1] Specifically, we use a Java optimization and analysis framework, Soot [52], to extract call graphs and FlowDroid [6] to ensure contexts and flows are preserved.

To better clarify the different steps involved in our system, we employ a "running example," using a real-world malware sample. Specifically, Fig. 2 lists a class extracted from the decompiled apk of malware disguised as a memory booster app (with package name com.g.o.speed.memboost), which executes commands (rm, chmod, etc.) as root.[2] To ease presentation, we focus on the portion of the code executed in the try/catch block. The resulting call graph of the try/catch block is shown in Fig. 3. Note that, for simplicity, we omit calls for object initialization, return types and parameters, as well as implicit calls in a method. Additional calls that are invoked when getShell(true) is called are not shown, except for the add() method that is directly called by the program code, as shown in Fig. 2.

C. Sequence Extraction

Next, we extract the sequences of API calls from the call graph. Since MaMaDroid uses static analysis, the graph obtained from Soot represents the sequence of functions that are potentially called by the program. However, each execution of the app could take a specific *branch* of the graph and only execute a subset of the calls. For instance, when running the code in Fig. 2 multiple times, the Execute method could be followed by different calls, e.g., getShell() in the try block only or getShell() and then getMessage() in the catch block.

[1]The standard Android archive file format containing all files, including the Java bytecode, making up the app.

[2]https://www.hackread.com/ghost-push-android-malware/

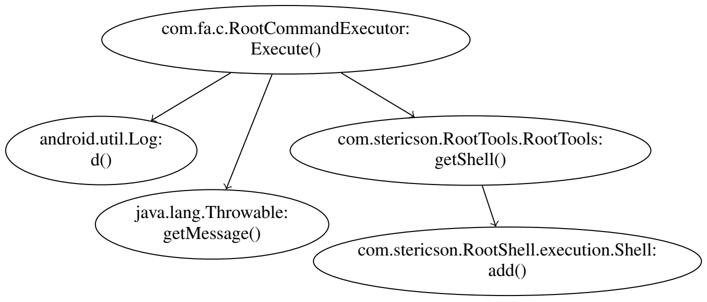

com.fa.c.RootCommandExecutor:
Execute()
android.util.Log:
d()
com.stericson.RootTools.RootTools:
getShell()
java.lang.Throwable:
getMessage()
com.stericson.RootShell.execution.Shell:
add()

Fig. 3: Call graph of the API calls in the try/catch block of Fig. 2. (Return types and parameters are omitted to ease presentation.)

In this phase, MaMaDroid operates as follows. First, it identifies a set of entry nodes in the call graph, i.e., nodes with no incoming edges (for example, the Execute method in the snippet from Fig. 2 is the entry node if there is no incoming edge from any other call in the app). Then, it enumerates the paths reachable from each entry node. The sets of all paths identified during this phase constitutes the sequences of API calls which will be used to build a Markov chain behavioral model and to extract features (see Section II-D).

Abstracting Calls to Families/Packages. Rather than analyzing raw API calls, we build MaMaDroid to work at a higher level, and operate in one of two modes by abstracting each call to either its package or family. This allows the system to be resilient to API changes and achieve scalability. In fact, our experiments, presented in Section III, show that, from a dataset of 44K apps, we extract more than 10 million unique API calls, which would result in a very large number of nodes, with the corresponding graphs (and feature vectors) being quite sparse. Since as we will see the number of features used by MaMaDroid is the square of the number of nodes, having more than 10 million nodes would result in an impractical computational cost.

When operating in package mode, we abstract an API call to its package name using the list of Android packages[3], which as of API level 24 (the current version as of September 2016) includes 243 packages, as well as 95 from the Google API.[4] Moreover, we abstract developer-defined packages (e.g., com.stericson.roottools) as well as obfuscated ones (e.g. com.fa.a.b.d), respectively, as `self-defined` and `obfuscated`. Note that we label an API call's package as obfuscated if we cannot tell what its class implements, extends, or inherits, due to identifier mangling [47]. When operating in family mode, we abstract to nine possible families, i.e., `android`, `google`, `java`, `javax`, `xml`, `apache`, `junit`, `json`, `dom`, which correspond to the android.*, com.google.*, java.*, javax.*, org.xml.*, org.apache.*, junit.*, org.json, and org.w3c.dom.* packages. Again, API calls from developer-defined and obfuscated packages are abstracted to families labeled as `self-defined` and `obfuscated`, respectively. Overall, there are 340 (243+95+2) possible packages and 11 (9+2) families. In Fig. 4, we show the sequence of API calls obtained from the call graph in Fig. 3. We also report, in square brackets, the family and the package to which the call is abstracted.

[3]https://developer.android.com/reference/packages.html

[4]https://developers.google.com/android/reference/packages

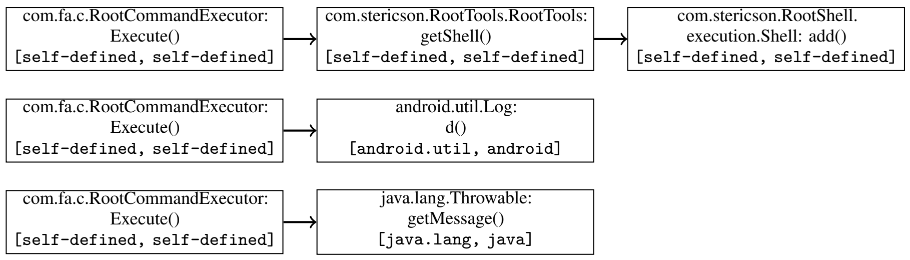

com.fa.c.RootCommandExecutor: Execute() [self-defined, self-defined]
com.stericson.RootTools.RootTools: getShell() [self-defined, self-defined]
com.stericson.RootShell. execution.Shell: add() [self-defined, self-defined]
com.fa.c.RootCommandExecutor: Execute() [self-defined, self-defined]
android.util.Log: d() [android.util, android]
com.fa.c.RootCommandExecutor: Execute() [self-defined, self-defined]
java.lang.Throwable: getMessage() [java.lang, java]

Fig. 4: Sequence of API calls extracted from the call graphs in Fig. 3, with the corresponding package/family abstraction in square brackets.

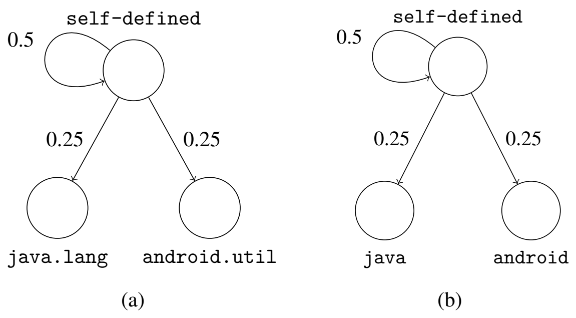


Fig. 5: Markov chains originating from the call sequence example in Section II-C when using packages (a) or families (b).

D. Markov-chain Based Modeling

Next, MAMADROID builds feature vectors, used for classification, based on the Markov chains representing the sequences of extracted API calls for an app. Before discussing this in detail, we review the basic concepts of Markov chains.

Markov chains are memoryless models where the probability of transitioning from a state to another only depends on the current state [39]. Markov chains are often represented as a set of nodes, each corresponding to a different state, and a set of edges connecting one node to another labeled with the probability of that transition. The sum of all probabilities associated to all edges from any node (including, if present, an edge going back to the node itself) is exactly 1.The set of possible states of the Markov chain is denoted as $\mathcal{S}$. If S_j and S_k are two connected states, P_{jk} denotes the probability of transition from S_j to S_k. P_{jk} is given by the number of occurrences (O_{jk}) of state S_k after state S_j, divided by O_{ji} for all states i in the chain, i.e., $P_{jk} = \frac{O_{jk}}{\sum_{i \in \mathcal{S}} O_{ji}}$.

Building the model. MAMADROID uses Markov chains to model app behavior, by evaluating every transition between calls. More specifically, for each app, MAMADROID takes as input the sequence of abstracted API calls of that app – i.e., packages or families, depending on the selected mode of operation – and builds a Markov chain where each package/family is a state and the transitions represent the probability of moving from one state to another. For each Markov chain, state S_0 is the entry point from which other calls are made in a sequence. As an example, Fig. 5 illustrates the two Markov chains built using packages and families, respectively, from the sequences reported in Fig. 4.

We argue that considering single transitions is more robust against attempts to evade detection by inserting useless API calls in order to deceive signature-based systems (see Section VI). In fact, MAMADROID considers all possible calls – i.e., all the branches originating from a node – in the Markov chain, so adding calls would not significantly change the probabilities of transitions between nodes (specifically, families or packages, depending on the operational mode) for each app.

Feature Extraction. Next, we use the probabilities of transitioning from one state (abstracted call) to another in the Markov chain as the feature vector of each app. States that are not present in a chain are represented as 0 in the feature vector. Also note that the vector derived from the Markov chain depends on the operational mode of MAMADROID. With families, there are 11 possible states, thus 121 possible transitions in each chain, while, when abstracting to packages, there are 340 states and 115,600 possible transitions.

We also apply Principal Component Analysis (PCA) [32], which performs feature selection by transforming the feature space into a new space made of components that are a linear combination of the original features. The first components contain as much variance (i.e., amount of information) as possible. The variance is given as percentage of the total amount of information of the original feature space. We apply PCA to the feature set in order to select the principal components, as PCA transforms the feature space into a smaller one where the variance is represented with as few components as possible, thus considerably reducing computation/memory complexity. Furthermore, the use of PCA could also improve the accuracy of the classification, by taking misleading features out of the feature space, i.e., those that make the classifier perform worse.

E. Classification

The last step is to perform classification, i.e., labeling apps as either benign or malware. To this end, we test MAMADROID using different classification algorithms: Random Forests [9], 1-Nearest Neighbor (1-NN) [22], 3-Nearest Neighbor (3-NN) [22], and Support Vector Machines (SVM) [29]. Each model is trained using the feature vector obtained from the apps in a training sample. Results are presented and discussed in Section IV, and have been validated by using 10-fold cross validation.

Also note that, due to the different number of features used in family/package modes, we use two distinct configurations for the Random Forests algorithm. Specifically, when abstracting to families, we use 51 trees with maximum depth 8, while, with packages, we use 101 trees of maximum depth 64. To tune Random Forests we followed the methodology applied in [7].

Category	Name	Date Range	#Samples	#Samples (API Calls)	#Samples (Call Graph)
Benign	oldbenign	Apr 2013 – Nov 2013	5,879	5,837	5,572
	newbenign	Mar 2016 – Mar 2016	2,568	2,565	2,465
		Total Benign:	*8,447*	*8,402*	*8,037*
Malware	drebin	Oct 2010 – Aug 2012	5,560	5,546	5,538
	2013	Jan 2013 – Jun 2013	6,228	6,146	6,123
	2014	Jun 2013 – Mar 2014	15,417	14,866	14,827
	2015	Jan 2015 – Jun 2015	5,314	5,161	4,725
	2016	Jan 2016 – May 2016	2,974	2,802	2,657
		Total Malware:	*35,493*	*34,521*	*33,870*

TABLE I: Overview of the datasets used in our experiments.

III. DATASETS

In this section, we introduce the datasets used in the evaluation of MAMADROID (presented later in Section IV), which include 43,940 apk files – 8,447 benign and 35,493 malware samples. We include a mix of older and newer apps, ranging from October 2010 to May 2016, as we aim to verify that MAMADROID is robust to changes in Android malware samples as well as APIs. To the best of our knowledge, we are leveraging the largest dataset of malware samples ever used in a research paper on Android malware detection.

Benign Samples. Our benign datasets consist of two sets of samples: (1) one, which we denote as oldbenign, includes 5,879 apps collected by PlayDrone [55] between April and November 2013, and published on the Internet Archive[5] on August 7, 2014; and (2) another, newbenign, obtained by downloading the top 100 apps in each of the 29 categories on the Google Play store[6] as of March 7, 2016, using the googleplay-api tool.[7] Due to errors encountered while downloading some apps, we have actually obtained 2,843 out of 2,900 apps. Note that 275 of these belong to more than one category, therefore, the newbenign dataset ultimately includes 2,568 unique apps.

Android Malware Samples. The set of malware samples includes apps that were used to test DREBIN [5], dating back to October 2010 – August 2012 (5,560), which we denote as drebin, as well as more recent ones that have been uploaded on the VirusShare[8] site over the years. Specifically, we gather from VirusShare, respectively, 6,228, 15,417, 5,314, and 2,974 samples from 2013, 2014, 2015, and 2016. We consider each of these datasets separately for our analysis.

API Calls and Call Graphs. For each app in our datasets, we extract the list of API calls, using Androguard[9], since, as explained in Section IV-E, these constitute the features used by DROIDAPIMINER [2], against which we compare our system. Due to Androguard failing to decompress some of the apks, bad CRC-32 redundancy checks, and errors during unpacking, we are not able to extract the API calls for all the samples, but only for 42,923 (8,402 benign, 34,521 malware) out of the 43,940 apps (8,447 benign, 35,493 malware) in our datasets. Also, to extract the call graph of each apk, we use Soot. Note that for some of the larger apks, Soot requires a non-negligible amount of memory to extract the call graph, so we allocate 16GB of RAM to the Java VM heap space. We find that for 2,027 (410 benign + 1,617 malware) samples, Soot is not able to complete the extraction due to it failing to apply the jb phase as well as reporting an error in opening some zip files (i.e., the apk). The jb phase is used by Soot to transform Java bytecode into jimple intermediate representation (the primary IR of Soot) for optimization purposes. Therefore, we exclude these apps in our evaluation and discuss this limitation further in Section V-C. In Table I, we provide a summary of our seven datasets, reporting the total number of samples per dataset, as well as those for which we are able to extract the API calls (second-to-last column) and the call graphs (last column).

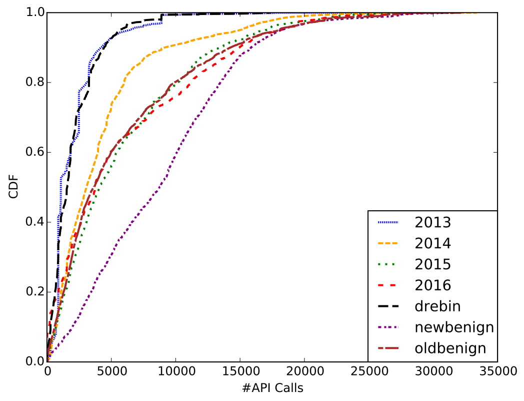
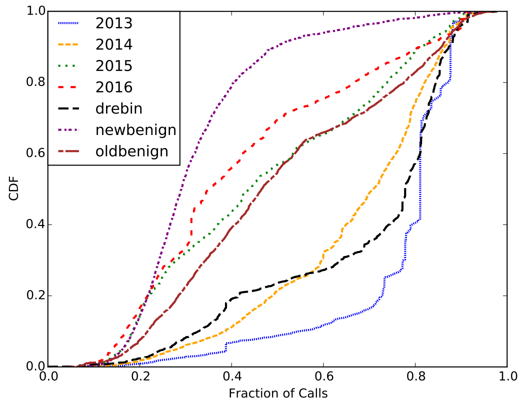


Fig. 6: CDF of the number of API calls in different apps in each dataset.

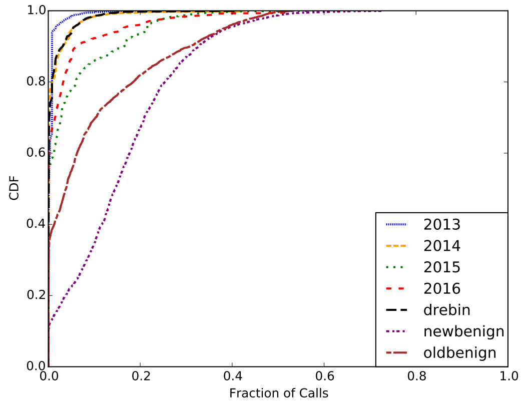
Characterization of the Datasets. Aiming to shed light on the evolution of API calls in Android apps, we also performed some measurements over our datasets. In Fig. 6, we plot the Cumulative Distribution Function (CDF) of the number of unique API calls in the apps in different datasets, highlighting that newer apps, both benign and malicious, are using more API calls overall than older apps. This indicates that as time goes by, Android apps become more complex. When looking at the fraction of API calls belonging to specific families, we discover some interesting aspects of Android apps developed in different years. In particular, we notice that API calls to the android family become less prominent as time passes (Fig. 7(a)), both in benign and malicious datasets, while google calls become more common in newer apps (Fig. 7(b)).

In general, we conclude that benign and malicious apps show the same evolutionary trends over the years. Malware, however, appears to reach the same characteristics (in terms of level of complexity and fraction of API calls from certain families) as legitimate apps with a few years of delay.

[5]https://archive.org/details/playdrone-apk-e8
[6]https://play.google.com/store
[7]https://github.com/egirault/googleplay-api
[8]https://virusshare.com/
[9]https://github.com/androguard/androguard

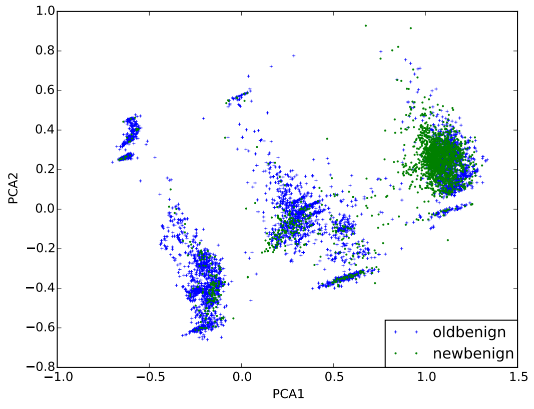


(a) `android`

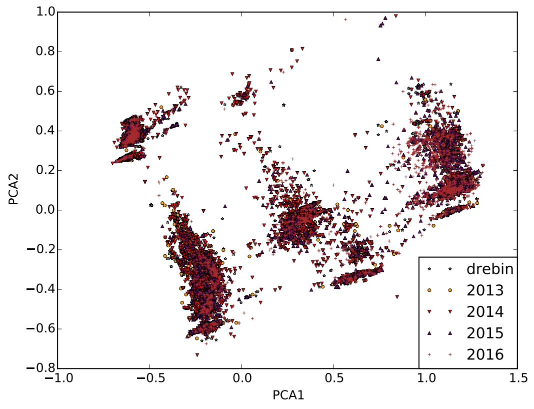


(b) `google`

Fig. 7: CDFs of the percentage of `android` and `google` family calls in different apps in each dataset.



(a) benign



(b) malware

Fig. 8: Positions of benign vs malware samples in the feature space of the first two components of the PCA (family mode).

Principal Component Analysis. Finally, we apply PCA to select the two most important PCA components. We plot and compare the positions of the two components for benign (Fig. 8(a)) and malicious samples (Fig. 8(b)). As PCA combines the features into components, it maximizes the variance of the distribution of samples in these components, thus, plotting the positions of the samples in the components shows that benign apps tend to be located in different areas of the components space, depending on the dataset, while malware samples occupy similar areas but with different densities. These differences highlight a different behavior between benign and malicious samples, and these differences should also be found by the machine learning algorithms used for classification.

IV. Evaluation

We now present a detailed experimental evaluation of MaMaDroid. Using the datasets summarized in Table I, we perform four sets of experiments: (1) we analyze the accuracy of MaMaDroid's classification on benign and malicious samples developed around the same time; (2) we evaluate its robustness to the evolution of malware as well as of the Android framework by using older datasets for training and newer ones for testing (and vice-versa); (3) we measure MaMaDroid's runtime performance to assess its scalability; and, finally, (4) we compare against DroidAPIMiner [2], a malware detection system that relies on the frequency of API calls.

A. *Preliminaries*

When implementing MaMaDroid in family mode, we exclude the `json` and `dom` families because they are almost never used across all our datasets, and `junit`, which is primarily used for testing. In package mode, to avoid mislabeling when `self-defined` APIs have "android" in the name, we split the `android` package into its two classes, i.e., `android.R` and `android.Manifest`. Therefore, in family mode, there are 8 possible states, thus 64 features, whereas, in package mode, we have 341 states and 116,281 features (cf. Section II-D).

As discussed in Section II-E, we use four different machine learning algorithms for classification – namely, Random Forests [9], 1-NN [22], 3-NN [22], and SVM [29]. Since both accuracy and speed are worse with SVM than with the other three algorithms, we omit results obtained with SVM. To assess the accuracy of the classification, we use the standard F-measure metric, i.e.:

$$F = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

where precision $=$ TP/(TP+FP) and recall $=$ TP/(TP+FN). TP denotes the number of samples correctly classified as malicious, while FP an FN indicate, respectively, the number of samples mistakenly identified as malicious and benign.

Finally, note that all our experiments perform 10-fold cross validation using at least one malicious and one benign dataset from Table I. In other words, after merging the datasets, the

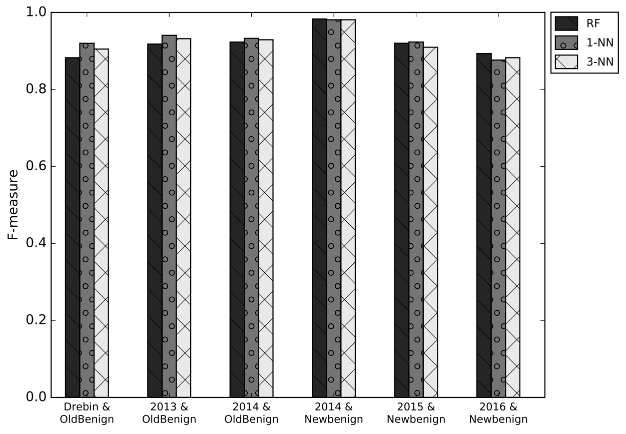


Fig. 9: F-measure of MAMADROID classification with datasets from the same year (family mode).

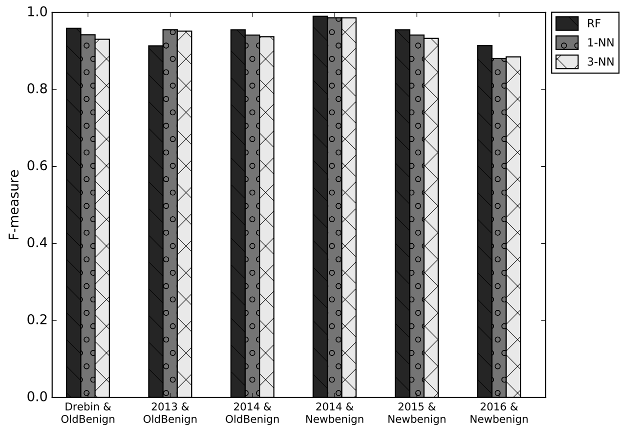


Fig. 10: F-measure of MAMADROID classification with datasets from the same year (package mode).

resulting set is shuffled and divided into ten equal-size random subsets. Classification is then performed ten times using nine subsets for training and one for testing, and results are averaged out over the ten experiments.

B. Detection Performance

We start our evaluation by measuring how well MAMADROID detects malware by training and testing using samples that are developed around the same time. To this end, we perform 10-fold cross validations on the combined dataset composed of a benign set and a malicious one. Table II provides an overview of the detection results achieved by MAMADROID on each combined dataset, in the two modes of operation, both with PCA features and without. The reported F-measure, precision, and recall scores are the ones obtained with Random Forest, which generally performs better than 1-NN and 3-NN.

Family mode. In Fig. 9, we report the F-measure when operating in family mode for Random Forests, 1-NN and 3-NN. The F-measure is always at least 88% with Random Forests, and, when tested on the 2014 (malicious) dataset, it reaches 98%. With some datasets, MAMADROID performs slightly better than with others. For instance, with the 2014 malware dataset, we obtain an F-measure of 92% when using the `oldbenign` dataset and 98% with `newbenign`. In general, lower F-measures are due to increased false positives since recall is always above 95%, while precision might be lower, also due to the fact that malware datasets are larger than the benign sets. We believe that this follows the evolutionary trend discussed in Section III: while both benign and malicious apps become more complex as time passes, when a new benign app is developed, it is still possible to use old classes or re-use code from previous versions and this might cause them to be more similar to old malware samples. This would result in false positives by MAMADROID. In general, MAMADROID performs better when the different characteristics of malicious and benign training and test sets are more predominant, which corresponds to datasets occupying different positions of the feature space.

Package mode. When MAMADROID runs in package mode, the classification performance improves, ranging from 92% F-measure with 2016 and `newbenign` to 99% with 2014 and `newbenign`, using Random Forests. Fig. 10 reports the F-measure of the 10-fold cross validation experiments using Random Forests, 1-NN, and 3-NN (in package mode). The former generally provide better results also in this case.

With some datasets, the difference in performance between the two modes of operation is more noticeable: with `drebin` and `oldbenign`, and using Random Forests, we get 95% F-measure in package mode compared to 88% in family mode. These differences are caused by a lower number of false positives in package mode. Recall remains high, resulting in a more balanced system overall. In general, abstracting to packages rather than families provides better results as the increased granularity enables identifying more differences between benign and malicious apps. On the other hand, however, this likely reduces the efficiency of the system, as many of the states deriving from the abstraction are used a only few times. The differences in time performance between the two modes are analyzed in details in Section IV-F.

Using PCA. As discussed in Section II-D, PCA transforms large feature spaces into smaller ones, thus it can be useful to significantly reduce computation and, above all, memory complexities of the classification task. When operating in package mode, PCA is particularly beneficial, since MAMADROID originally has to operate over 116,281 features. Therefore, we compare results obtained using PCA by fixing the number of components to 10 and checking the quantity of variance included in them. In package mode, we observe that only 67% of the variance is taken into account by the 10 most important PCA components, whereas, in family mode, at least 91% of the variance is included by the 10 PCA Components.

As shown in Table II, the F-measure obtained using Random Forests and the PCA components sets derived from the family and package features is only slightly lower (up to 4%) than using the full feature set. We note that lower F-measures are caused by a uniform decrease in both precision and recall.

C. Detection Over Time

As Android evolves over the years, so do the characteristics of both benign and malicious apps. Such evolution must be taken into account when evaluating Android malware detection

Dataset / Mode	drebin & oldbenign			2013 & oldbenign			2014 & oldbenign			2014 & newbenign			2015 & newbenign			2016 & newbenign		
	[Precision, Recall, **F-measure**]																	
Family	0.82	0.95	**0.88**	0.91	0.93	**0.92**	0.88	0.97	**0.92**	0.97	0.99	**0.98**	0.90	0.94	**0.92**	0.87	0.92	**0.89**
Package	0.95	0.97	**0.96**	0.98	0.96	**0.97**	0.93	0.98	**0.96**	0.98	1.00	**0.99**	0.93	0.98	**0.96**	0.91	0.91	**0.91**
Family (PCA)	0.83	0.93	**0.87**	0.93	0.90	**0.91**	0.86	0.94	**0.90**	0.96	0.99	**0.97**	0.87	0.93	**0.90**	0.86	0.87	**0.87**
Package (PCA)	0.93	0.95	**0.94**	0.97	0.94	**0.95**	0.92	0.96	**0.94**	0.98	1.00	**0.99**	0.92	0.97	**0.94**	0.88	0.90	**0.89**

TABLE II: F-measure, precision, and recall obtained by MAMADROID, using Random Forests, on various dataset combinations with different modes of operation, with and without PCA.

systems, since their accuracy might significantly be affected as newer APIs are released and/or as malicious developers modify their strategies in order to avoid detection. Evaluating this aspect constitutes one of our research questions, and one of the reasons why our datasets span across multiple years (2010–2016).

As discussed in Section II-B, MAMADROID relies on the sequence of API calls extracted from the call graphs and abstracted at either the package or the family level. Therefore, it is less susceptible to changes in the Android API than other classification systems such as DROIDAPIMINER [2] and DREBIN [5]. Since these rely on the use, or the frequency, of certain API calls to classify malware vs benign samples, they need to be retrained following new API releases. On the contrary, retraining is not needed as often with MAMADROID, since families and packages represent more abstract functionalities that change less over time. Consider, for instance, the `android.os.health` package: released with API level 24, it contains a set of classes helping developers track and monitor system resources.[10] Classification systems built before this release – as in the case of DROIDAPIMINER [2] (released in 2013, when Android API was up to level 20) – need to be retrained if this package is more frequently used by malicious apps than benign apps, while MAMADROID only needs to add a new state to its Markov chain when operating in package mode, while no additional state is required when operating in family mode.

To verify this hypothesis, we test MAMADROID using older samples as training sets and newer ones as test sets. Fig. 11 reports the F-measure of the classification in this setting, with MAMADROID operating in family mode. The x-axis reports the difference in years between training and test data. We obtain 86% F-measure when we classify apps one year older than the samples on which we train. Classification is still relatively accurate, at 75%, even after two years. Then, from Fig. 12, we observe that the F-measure does not significantly change when operating in package mode. Both modes of operations are affected by one particular condition, already discussed in Section III: in our models, benign datasets seem to "anticipate" malicious ones by 1–2 years in the way they use certain API calls. As a result, we notice a drop in accuracy when classifying future samples and using `drebin` (with samples from 2010 to 2012) or 2013 as the malicious training set and `oldbenign` (late 2013/early 2014) as the benign training set. More specifically, we observe that MAMADROID correctly detects benign apps, while it starts missing true positives and increasing false negatives — i.e., achieving lower recall.

We also set to verify whether older malware samples can still be detected by the system—if not, this would obviously become

[10]https://developer.android.com/reference/android/os/health/package-summary.html

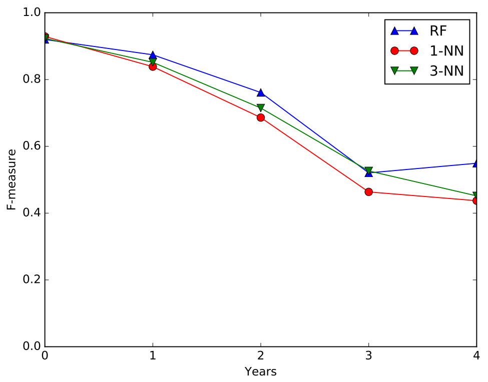


Fig. 11: F-measure of MAMADROID classification using older samples for training and newer for testing (family mode).

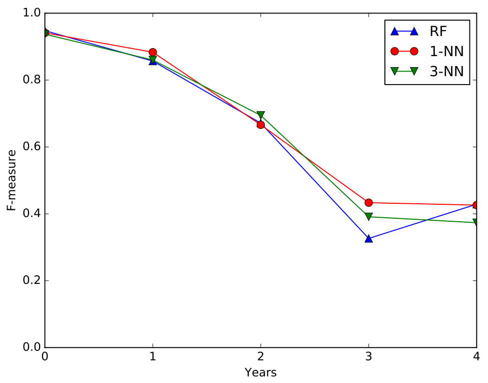


Fig. 12: F-measure of MAMADROID classification using older samples for training and newer for testing (package mode).

vulnerable to older (and possibly popular) attacks. Therefore, we also perform the "opposite" experiment, i.e., training MAMADROID with newer datasets, and checking whether it is able to detect malware developed years before. Specifically, Fig. 13 and 14 report results when training MAMADROID with samples from a given year, and testing it with others that are up to 4 years older: MAMADROID retains similar F-measure scores over the years. Specifically, in family mode, it varies from 93% to 96%, whereas, in package mode, from 95% to 97% with the oldest samples.

D. Case Studies of False Positives and Negatives

The experiment analysis presented above show that MAMADROID detects Android malware with high accuracy. As in any detection system, however, the system makes a small number of incorrect classifications, incurring some false positives and false negatives. Next, we discuss a few case studies aiming to better understand these misclassifications. We focus on the experiments with newer datasets, i.e., 2016 and

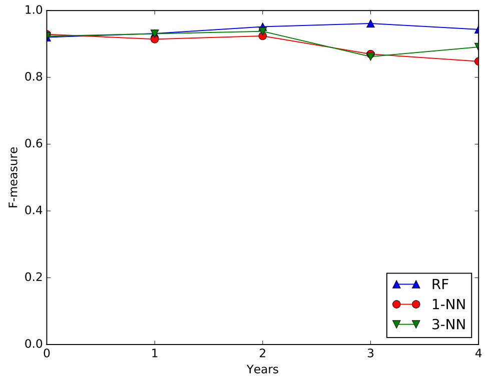


Fig. 13: F-measure of MAMADROID classification using newer samples for training and older for testing (family mode).

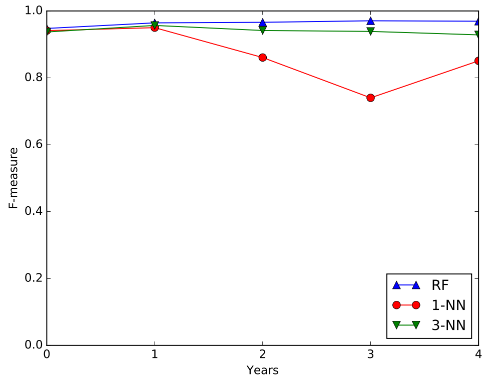


Fig. 14: F-measure of MAMADROID classification using newer samples for training and older for testing (package mode).

`newbenign`.

False Positives. We analyze the manifest of the 164 apps mistakenly detected as malware by MAMADROID, finding that most of them use "dangerous" permissions [4]. In particular, 67% of the apps write to external storage, 32% read the phone state, and 21% access the device's fine location. We further analyzed apps (5%) that use the READ_SMS and SEND_SMS permissions, i.e., even though they are not SMS-related apps, they can read and send SMSs as part of the services they provide to users. In particular, a *"in case of emergency"* app is able to send messages to several contacts from its database (possibly added by the user), which is a typical behavior of Android malware in our dataset, ultimately leading MAMADROID to flag it as malicious.

False Negatives. We also check the 114 malware samples missed by MAMADROID when operating in family mode, using VirusTotal.[11] We find that 18% of the false negatives are actually not classified as malware by any of the antivirus engines used by VirusTotal, suggesting that these are actually legitimate apps mistakenly included in the VirusShare dataset. 45% of MAMADROID's false negatives are *adware*, typically, repackaged apps in which the advertisement library has been substituted with a third-party one, which creates a monetary profit for the developers. Since they are not performing any clearly malicious activity, MAMADROID is unable to identify them as malware. Finally, we find that 16% of the false negatives reported by MAMADROID are samples sending text messages or starting calls to premium services. We also do a similar analysis of false negatives when abstracting to packages (74 samples), with similar results: there a few more adware samples (53%), but similar percentages for potentially benign apps (15%) and samples sending SMSs or placing calls (11%).

In conclusion, we find that MAMADROID's sporadic misclassifications are typically due to benign apps behaving similarly to malware, malware that do not perform clearly-malicious activities, or mistakes in the ground truth labeling.

E. MAMADROID *vs* DROIDAPIMINER

We also compare the performance of MAMADROID to previous work using API features for Android malware classification. Specifically, we compare to DROIDAPIMINER [2], because: (i) it uses API calls and its parameters to perform classification; (ii) it reports high true positive rate (up to 97.8%) on almost 4K malware samples obtained from McAfee and GENOME [66], and 16K benign samples; and (iii) its source code has been made available to us by the authors.

In DROIDAPIMINER, permissions that are requested more frequently by malware samples than by benign apps are used to perform a baseline classification. Since there are legitimate situations where a non-malicious app needs permissions tagged as dangerous, DROIDAPIMINER also applies frequency analysis on the list of API calls, specifically, using the 169 most frequent API calls in the malware samples (occurring at least 6% more in malware than benign samples) —leading to a reported 83% precision. Finally, data flow analysis is applied on the API calls that are frequent in both benign and malicious samples, but do not occur by at least, 6% more in the malware set. Using the top 60 parameters, the 169 most frequent calls change, and authors report a precision of 97.8%.

After obtaining DROIDAPIMINER's source code, as well as a list of packages used for feature refinement, we re-implement the system by modifying the code in order to reflect recent changes in Androguard (used by DROIDAPIMINER for API call extraction), extract the API calls for all apps in the datasets listed in Table I, and perform a frequency analysis on the calls. Androguard fails to extract calls for about 2% (1,017) of apps in our datasets as a result of bad CRC-32 redundancy checks and error in unpacking, thus DROIDAPIMINER is evaluated over the samples in the second-to-last column of Table I. We also implement classification, which is missing from the code provided by the authors, using k-NN (with k=3) since it achieves the best results according to the paper. We use 2/3 of the dataset for training and 1/3 for testing as implemented by the authors [2]. A summary of the resulting F-measures obtained using different training and test sets is presented in Table III.

We set up a number of experiments to thoroughly compare DROIDAPIMINER to MAMADROID. First, we set up three experiments in which we train DROIDAPIMINER using a dataset composed of `oldbenign` combined with one of the three oldest malware datasets each (`drebin`, 2013, and 2014), and testing on all malware datasets. With this configuration, the best result (with 2014 and `oldbenign` as training sets) amounts to 62% F-measure when tested on the same dataset.

[11] https://www.virustotal.com

	Testing Sets									
	drebin & oldbenign		2013 & oldbenign		2014 & oldbenign		2015 & oldbenign		2016 & oldbenign	
Training Sets	[2]	Our Work	[2]	Our Work	[2]	Our Work	[2]	Our Work	[2]	Our Work
drebin & oldbenign	0.32	**0.96**	0.35	**0.96**	0.34	**0.79**	0.30	**0.42**	0.33	**0.43**
2013 & oldbenign	0.33	**0.93**	0.36	**0.97**	0.35	**0.74**	0.31	**0.36**	**0.33**	0.29
2014 & oldbenign	0.36	**0.92**	0.39	**0.93**	0.62	**0.95**	0.33	**0.79**	0.37	**0.78**
	drebin & newbenign		2013 & newbenign		2014 & newbenign		2015 & newbenign		2016 & newbenign	
Training Sets	[2]	Our Work	[2]	Our Work	[2]	Our Work	[2]	Our Work	[2]	Our Work
2014 & newbenign	0.76	**0.99**	0.75	**0.99**	0.92	**0.99**	0.67	**0.89**	0.65	**0.83**
2015 & newbenign	0.68	**0.98**	0.68	**0.98**	0.69	**0.99**	0.77	**0.95**	0.65	**0.90**
2016 & newbenign	0.33	**0.97**	0.35	**0.97**	0.36	**0.99**	0.34	**0.93**	0.36	**0.92**

TABLE III: Classification performance of DroidAPIMiner [2] vs MaMaDroid (our work).

The F-measure drops to 33% and 39%, respectively, when tested on samples one year into the future and past. If we use the same configurations in MaMaDroid, in package mode, we obtain up to 97% F-measure (using 2013 and oldbenign as training sets), dropping to 74% and 93%, respectively, one year into the future and into the past. For the datasets where DroidAPIMiner achieves its best result (i.e., 2014 and oldbenign), MaMaDroid achieves an F-measure of 95%, which drops to respectively, 79% and 93% one year into the future and the past. The F-measure is stable even two years into the future and the past at 78% and 92%, respectively.

As a second set of experiments, we train DroidAPIMiner using a dataset composed of newbenign combined with one of the three most recent malware datasets each (2014, 2015, and 2016). Again, we test DroidAPIMiner on all malware datasets. The best result is obtained with the dataset (2014 and newbenign) used for both testing and training, yielding a F-measure of 92%, which drops to 67% and 75% one year into the future and past respectively. Likewise, we use the same datasets for MaMaDroid, with the best results achieved on the same dataset as DroidAPIMiner. In package mode, MaMaDroid achieves an F-measure of 99%, which is maintained more than two years into the past, but drops to respectively, 89% and 83% one and two years into the future.

As summarized in Table III, MaMaDroid achieves significantly higher performance than DroidAPIMiner in all but one experiment, with the F-measure being at least 79% even after two years into the future or the past when datasets from 2014 or later are used for training. Note that there is only one setting in which DroidAPIMiner performs slightly better than MaMaDroid: this occurs when the malicious training set is much older than the malicious test set. Specifically, MaMaDroid presents low recall in this case: as discussed, MaMaDroid's classification performs much better when the training set is not more than two years older than the test set.

F. Runtime Performance

We envision MaMaDroid to be integrated in offline detection systems, e.g., run by Google Play. Recall that MaMaDroid consists of different phases, so in the following, we review the computational overhead incurred by each of them, aiming to assess the feasibility of real-world deployment. We run our experiments on a desktop equipped with an 40-core 2.30GHz CPU and 128GB of RAM, but only use one core and allocate 16GB of RAM for evaluation.

MaMaDroid's first step involves extracting the call graph from an apk and the complexity of this task varies significantly across apps. On average, it takes 9.2s±14 (min 0.02s, max 13m) to complete for samples in our malware sets. Benign apps usually yield larger call graphs, and the average time to extract them is 25.4s±63 (min 0.06s, max 18m) per app. Note that we do not include in our evaluation apps for which we could not successfully extract the call graph.

Next, we measure the time needed to extract call sequences while abstracting to families or packages, depending on MaMaDroid's mode of operation. In family mode, this phase completes in about 1.3s on average (and at most 11.0s) with both benign and malicious samples. Abstracting to packages takes slightly longer, due to the use of 341 packages in MaMaDroid. On average, this extraction takes 1.67s±3.1 for malicious apps and 1.73s±3.2 for benign samples. As it can be seen, the call sequence extraction in package mode does not take significantly more than in family mode.

MaMaDroid's third step includes Markov chain modeling and feature vector extraction. This phase is fast regardless of the mode of operation and datasets used. Specifically, with malicious samples, it takes on average 0.2s±0.3 and 2.5s±3.2 (and at most 2.4s and 22.1s), respectively, with families and packages, whereas, with benign samples, averages rise to 0.6s±0.3 and 6.7s±3.8 (at most 1.7s and 18.4s).

Finally, the last step involves classification, and performance depends on both the machine learning algorithm employed and the mode of operation. More specifically, running times are affected by the number of features for the app to be classified, and not by the initial dimension of the call graph, or by whether the app is benign or malicious. Regardless, in family mode, Random Forests, 1-NN, and 3-NN all take less than 0.01s. With packages, it takes, respectively, 0.65s, 1.05s, and 0.007s per app with 1-NN, 3-NN, Random Forests.

Overall, when operating in family mode, malware and benign samples take on average, 10.7s and 27.3s respectively to complete the entire process, from call graph extraction to classification. Whereas, in package mode, the average completion times for malware and benign samples are 13.37s and 33.83s respectively. In both modes of operation, time is mostly (> 80%) spent on call graph extraction.

We also evaluate the runtime performance of DroidAPIMiner [2]. Its first step, i.e., extracting API calls, takes 0.7s±1.5 (min 0.01s, max 28.4s) per app in our malware datasets. Whereas, it takes on average 13.2s±22.2 (min 0.01s, max 222s) per benign app. In the second phase, i.e., frequency and data flow analysis, it takes, on average, 4.2s per app. Finally, classification using 3-NN is very fast: 0.002s on average. Therefore, in total, DroidAPIMiner takes respectively, 17.4s and 4.9s for a complete execution on one app from our benign and malware datasets, which while faster than MAMADROID, achieves significantly lower accuracy.

In conclusion, our experiments show that our prototype implementation of MAMADROID is scalable enough to be deployed. Assuming that, everyday, a number of apps in the order of 10,000 are submitted to Google Play, and using the average execution time of benign samples in family (27.3s) and package (33.83s) modes, we estimate that it would take less than an hour and a half to complete execution of all apps submitted daily in both modes, with just 64 cores. Note that we could not find accurate statistics reporting the number of apps submitted everyday, but only the total number of apps on Google Play.[12] On average, this number increases of a couple of thousands per day, and although we do not know how many apps are removed, we believe 10,000 apps submitted every day is likely an upper bound.

V. DISCUSSION

We now discuss the implications of our results with respect to the feasibility of modeling app behavior using static analysis and Markov chains, discuss possible evasion techniques, and highlight some limitations of our approach.

A. *Lessons Learned*

Our work yields important insights around the use of API calls in malicious apps, showing that, by modeling the sequence of API calls made by an app as a Markov chain, we can successfully capture the behavioral model of that app. This allows MAMADROID to obtain high accuracy overall, as well as to retain it over the years, which is crucial due to the continuous evolution of the Android ecosystem.

As discussed in Section III, the use of API calls changes over time, and in different ways across malicious and benign samples. From our newer datasets, which include samples up to Spring 2016 (API level 23), we observe that newer APIs introduce more packages, classes, and methods, while also deprecating some. Fig. 6, 7(a), and 7(b) show that benign apps are using more calls than malicious ones developed around the same time. We also notice an interesting trend in the use of Android and Google APIs: malicious apps follow the same trend as benign apps in the way they adopt certain APIs, but with a delay of some years. This might be a side effect of Android malware authors' tendency to repackage benign apps, adding their malicious functionalities onto them.

Given the frequent changes in the Android framework and the continuous evolution of malware, systems like DROIDAPIMINER [2] – being dependent on the presence or the use of certain API calls – become increasingly less effective with time. As shown in Table III, malware that uses API calls released after those used by samples in the training set cannot be identified by these systems. On the contrary, as shown in Fig. 11 and 12, MAMADROID detects malware samples that are *1 year* newer than the training set obtaining an 86% F-measure (as opposed to 46% with DROIDAPIMINER). After 2 years, the value is still at 75% (42% with DROIDAPIMINER), dropping to 51% after 4 years.

[12]http://www.appbrain.com/stats/number-of-android-apps

We argue that the effectiveness of MAMADROID's classification remains relatively high "over the years" owing to Markov models capturing app behavior. These models tend to be more robust to malware evolution because abstracting to families or packages makes the system less susceptible to the introduction of new API calls. Abstraction allows MAMADROID to capture newer classes/methods added to the API, since these are abstracted to already-known families or packages. In case newer packages are added to the API, and these packages start being used by malware, MAMADROID only requires adding a new state to the Markov chains, and probabilities of a transition from a state to this new state in old apps would be 0. Adding only a few nodes does not likely alter the probabilities of the other 341 nodes, thus, two apps created with the same purpose will not strongly differ in API calls usage if they are developed using almost consecutive API levels.

We also observe that abstracting to packages provides a slightly better tradeoff than families. In family mode, the system is lighter and faster, and actually performs better when there are more than two years between training and test set samples However, even though both modes of operation effectively detect malware, abstracting to packages yields better results overall. Nonetheless, this does not imply that less abstraction is always better: in fact, a system that is too granular, besides incurring untenable complexity, would likely create Markov models with low-probability transitions, ultimately resulting in less accurate classification. We also highlight that applying PCA is a good strategy to preserve high accuracy and at the same time reducing complexity.

B. *Evasion*

Next, we discuss possible evasion techniques and how they can be addressed. One straightforward evasion approach could be to repackage a benign app with small snippets of malicious code added to a few classes. However, it is difficult to embed malicious code in such a way that, at the same time, the resulting Markov chain looks similar to a benign one. For instance, our running example from Section II (malware posing as a memory booster app and executing unwanted commands as root) is correctly classified by MAMADROID; although most functionalities in this malware are the same as the original app, injected API calls generate some transitions in the Markov chain that are not typical of benign samples.

The opposite procedure – i.e., embedding portions of benign code into a malicious app – is also likely ineffective against MAMADROID, since, for each app, we derive the feature vector from the transition probability between calls over the entire app. In other words, a malware developer would have to embed benign code inside the malware in such a way that the overall sequence of calls yields similar transition probabilities as those in a benign app, but this is difficult to achieve because if the sequences of calls have to be different (otherwise there would be no attack), then the models will also be different.

An attacker could also try to create an app from scratch with a similar Markov chain to that of a benign app. Because this is derived from the sequence of abstracted API calls in the app, it is actually very difficult to create sequences resulting in Markov chains similar to benign apps while, at the same time, actually engaging in malicious behavior. Nonetheless, in future work, we plan to systematically analyze the feasibility of this strategy.

Moreover, attackers could try using reflection, dynamic code loading, or native code [42]. Because MaMaDroid uses static analysis, it fails to detect malicious code when it is loaded or determined at runtime. However, MaMaDroid can detect reflection when a method from the reflection package (`java.lang.reflect`) is executed. Therefore, we obtain the correct sequence of calls up to the invocation of the reflection call, which may be sufficient to distinguish between malware and benign apps. Similarly, MaMaDroid can detect the usage of class loaders and package contexts that can be used to load arbitrary code, but it is not able to model the code loaded; likewise, native code that is part of the app cannot be modeled, as it is not Java and is not processed by Soot. These limitations are not specific of MaMaDroid, but are a problem of static analysis in general, which can be mitigated by using MaMaDroid alongside dynamic analysis techniques.

Malware developers might also attempt to evade MaMaDroid by naming their self-defined packages in such a way that they look similar to that of the `android`, `java`, or `google` APIs, e.g., creating packages like java.lang.reflect.*malware* and java.lang.*malware*, aiming to confuse MaMaDroid into abstracting them to respectively, `java.lang.reflect` and `java.lang`. However, this is easily prevented by whitelisting the list of packages from `android`, `java`, or `google` APIs.

Another approach could be using dynamic dispatch so that a class X in package A is created to extend class Y in package B with static analysis reporting a call to root() defined in Y as X.root(), whereas, at runtime Y.root() is executed. This can be addressed, however, with a small increase in MaMaDroid's computational cost, by keeping track of self-defined classes that extend or implement classes in the recognized APIs, and abstract polymorphic functions of this self-defined class to the corresponding recognized package, while, at the same time, abstracting as self-defined overridden functions in the class.

Finally, identifier mangling and other forms of obfuscation could be used aiming to obfuscate code and hide malicious actions. However, since classes in the Android framework cannot be obfuscated by obfuscation tools, malware developers can only do so for self-defined classes. MaMaDroid labels obfuscated calls as `obfuscated` so, ultimately, these would be captured in the behavioral model (and the Markov chain) for the app. In our sample, we observe that benign apps use significantly less obfuscation than malicious apps, indicating that obfuscating a significant number of classes is not a good evasion strategy since this would likely make the sample more easily identifiable as malicious.

C. Limitations

MaMaDroid requires a sizable amount of memory in order to perform classification, when operating in package mode, working on more than 100,000 features per sample. The quantity of features, however, can be further reduced using feature selection algorithms such as PCA. As explained in Section IV when we use 10 components from the PCA the system performs almost as well as the one using all the features; however, using PCA comes with a much lower memory complexity in order to run the machine learning algorithms, because the number of dimensions of the features space where the classifier operates is remarkably reduced.

Soot [52], which we use to extract call graphs, fails to analyze some apks. In fact, we were not able to extract call graphs for a fraction (4.6%) of the apps in the original datasets due to scripts either failing to apply the `jb` phase, which is used to transform Java bytecode to the primary intermediate representation (i.e., jimple) of Soot or not able to open the apk. Even though this does not really affect the results of our evaluation, one could avoid it by using a different/custom intermediate representation for the analysis or use different tools to extract the call graphs.

In general, static analysis methodologies for malware detection on Android could fail to capture the runtime environment context, code that is executed more frequently, or other effects stemming from user input [5]. These limitations can be addressed using dynamic analysis, or by recording function calls on a device. Dynamic analysis observes the live performance of the samples, recording what activity is actually performed at runtime. Through dynamic analysis, it is also possible to provide inputs to the app and then analyze the reaction of the app to these inputs, going beyond static analysis limits. To this end, we plan to integrate dynamic analysis to build the models used by MaMaDroid as part of future work.

VI. Related Work

Over the past few years, Android security has attracted a wealth of work by the research community. In this section, we review (i) program analysis techniques focusing on general security properties of Android apps, and then (ii) systems that specifically target malware on Android.

A. Program Analysis

Previous work on program analysis applied to Android security has used both static and dynamic analysis. With the former, the program's code is decompiled in order to extract features without actually running the program, usually employing tools such as Dare [41] to obtain Java bytecode. The latter involves real-time execution of the program, typically in an emulated or protected environment.

Static analysis techniques include work by Felt et al. [21], who analyze API calls to identify over-privileged apps, while Kirin [20] is a system that examines permissions requested by apps to perform a lightweight certification, using a set of security rules that indicate whether or not the security configuration bundled with the app is safe. RiskRanker [28] aims to identify zero-day Android malware by assessing potential security risks caused by untrusted apps. It sifts through a large number of apps from Android markets and examines them to detect certain behaviors, such as encryption and dynamic code loading, which form malicious patterns and can be used to detect stealthy malware. Other methods, such as CHEX [37], use data flow analysis to automatically vet Android apps for vulnerabilities. Static analysis has also been applied to the detection of data leaks and malicious data flows from Android apps [6], [34], [35], [62].

DroidScope [59] and TaintDroid [19] monitor run-time app behavior in a protected environment to perform dynamic taint analysis. DroidScope performs dynamic taint analysis at the machine code level, while TaintDroid monitors how third-party apps access or manipulate users' personal data, aiming to detect sensitive data leaving the system. However, as it is unrealistic to deploy dynamic analysis techniques directly on users' devices, due to the overhead they introduce, these are typically used offline [45], [50], [67]. ParanoidAndroid [44] employs a virtual clone of the smartphone, running in parallel in the cloud and replaying activities of the device – however, even if minimal execution traces are actually sent to the cloud, this still takes a non-negligible toll on battery life.

Recently, hybrid systems like IntelliDroid [56] have also been proposed that use input generators, producing inputs specific to dynamic analysis tools. Other work combining static and dynamic analysis include [8], [25], [31], [58].

B. Android Malware Detection

A number of techniques have used *signatures* for Android malware detection. NetworkProfiler [18] generates network profiles for Android apps and extracts fingerprints based on such traces, while work in [12] obtains resource-based metrics (CPU, memory, storage, network) to distinguish malware activity from benign one. Chen et al. [15] extract statistical features, such as permissions and API calls, and extend their vectors to add dynamic behavior-based features. While their experiments show that their solution outperforms, in terms of accuracy, other antivirus systems, Chen et al. [15] indicate that the quality of their detection model critically depends on the availability of representative benign and malicious apps for training. Similarly, ScanMe Mobile [64] uses the Google Cloud Messaging Service (GCM) to perform static and dynamic analysis on apks found on the device's SD card.

The sequences of system calls have also been used to detect malware in both desktop and Android environments. Hofmeyr et al. [30] demonstrate that short sequences of system calls can be used as a signature to discriminate between normal and abnormal behavior of common UNIX programs. Signature-based methods, however, can be evaded using polymorphism and obfuscation, as well as by call re-ordering attacks [36], even though quantitative measures, such as similarity analysis, can be used to address some of these attacks [48]. MAMADROID inherits the spirit of these approaches, proposing a statistical method to model app behavior that is more robust against evasion attempts.

In the Android context, Canfora et al. [11] use the sequences of three system calls (extracted from the execution traces of apps under analysis) to detect malware. This approach models specific malware families, aiming to identify additional samples belonging to such families. In contrast, MAMADROID's goal is to detect previously-unseen malware, and we also show that our system can detect new malware families that even appear years after the system has been trained. In addition, using strict sequences of system or API calls can be easily evaded by malware authors who could add unnecessary calls to effectively evade detection. Conversely, MAMADROID builds a behavioral model of an Android app, which makes it robust to this type of evasion.

Dynamic analysis has also been applied to detect Android malware by using predefined scripts of common inputs that will be performed when the device is running. However, this might be inadequate due to the low probability of triggering malicious behavior, and can be side-stepped by knowledgeable adversaries, as suggested by Wong and Lie [56]. Other approaches include random fuzzing [38], [63] and concolic testing [3], [26]. Dynamic analysis can only detect malicious activities if the code exhibiting malicious behavior is actually running during the analysis. Moreover, according to [54], mobile malware authors often employ emulation or virtualization detection strategies to change malware behavior and eventually evade detection.

Aiming to complement static and dynamic analysis tools, machine learning techniques have also been applied to assist Android malware detection. Droidmat [57] uses API call tracing and manifest files to learn features for malware detection, while Gascon et al. [24] rely on embedded call graphs. DroidMiner [60] studies the program logic of sensitive Android/Java framework API functions and resources, and detects malicious behavior patterns. MAST [14] statically analyzes apps using features such as permissions, presence of native code, and intent filters and measures the correlation between multiple qualitative data. Crowdroid [10] relies on crowdsourcing to distinguish between malicious and benign apps by monitoring system calls. AppContext [61] models security-sensitive behavior, such as activation events or environmental attributes, and uses SVM to classify these behaviors, while RevealDroid [23] employs supervised learning and obfuscation-resilient methods targeting API usage and intent actions to identify their families.

DREBIN [5] automatically deduces detection patterns and identifies malicious software directly on the device, performing a broad static analysis. This is achieved by gathering numerous features from the manifest file as well as the app's source code (API calls, network addresses, permissions). Malevolent behavior is reflected in patterns and combinations of extracted features from the static analysis: for instance, the existence of both SEND_SMS permission and the android.hardware.telephony component in an app might indicate an attempt to send premium SMS messages, and this combination can eventually constitute a detection pattern.

In Section IV, we have already introduced, and compared against, DROIDAPIMINER [2]. This system relies on the top-169 API calls that are used more frequently in the malware than in the benign set, along with data flow analysis on calls that are frequent in both benign and malicious apps, but occur up to 6% more in the latter. As shown in our evaluation, using the most common calls observed during training requires constant retraining, due to the evolution of both malware and the Android API. On the contrary, MAMADROID can effectively model both benign and malicious Android apps, and perform an efficient classification on them. Compared to DROIDAPIMINER, our approach is more resilient to changes in the Android framework than DROIDAPIMINER, resulting in a less frequent need to re-train the classifier.

Overall, compared to state-of-the-art systems like DREBIN [5] and DROIDAPIMINER [2], MAMADROID is more generic and robust as its statistical modeling does not depend on specific app characteristics, but can actually be run on any app created for any Android API level.

Finally, also related to MAMADROID are Markov-chain based models for Android malware detection. Chen et al. [16] dynamically analyze system- and developer-defined actions from intent messages (used by app components to communicate with each other at runtime), and probabilistically estimate whether an app is performing benign or malicious actions at run time, but obtain low accuracy overall. Canfora et al. [13] use a Hidden Markov model (HMM) to identify malware samples belonging to previously observed malware families, whereas, MaMaDroid can detect previously unseen malware, not relying on specific malware families.

VII. Conclusion

This paper presented MaMaDroid, an Android malware detection system based on modeling the sequences of API calls as Markov chains. Our system is designed to operate in one of two modes, with different granularities, by abstracting API calls to either families or packages. We ran an extensive experimental evaluation using, to the best of our knowledge, the largest malware dataset ever analyzed in an Android malware detection research paper, and aiming at assessing both the accuracy of the classification (using F-measure, precision, and recall) and runtime performances. We showed that MaMaDroid effectively detects unknown malware samples developed earlier or around the same time as the samples on which it is trained (F-measure up to 99%). It also maintains good detection performance: one year after the model has been trained the F-measure value is 86%, and after two years it is 75%.

We compared MaMaDroid to DroidAPIMiner [2], a state-of-the-art system based on API calls frequently used by malware, showing that, not only does MaMaDroid outperforms DroidAPIMiner when trained and tested on the same datasets, but that it is also much more resilient over the years to changes in the Android API. Overall, our results demonstrate that the type of statistical behavioral models introduced by MaMaDroid are more robust than traditional techniques, highlighting how our work can form the basis of more advanced detection systems in the future. As part of future work, we plan to further investigate the resilience to possible evasion techniques, focusing on repackaged malicious apps as well as injection of API calls to maliciously alter Markov models. We also plan to explore the use of finer-grained abstractions as well as the possibility to seed the behavioral modeling performed by MaMaDroid with dynamic instead of static analysis. Due to the large size of the data, we have not made them readily available online but both the datasets and the feature vectors can be obtained upon request.

Acknowledgments. We wish to thank the anonymous reviewers for their feedback, our shepherd Amir Houmansadr for his help in improving our paper, and Yousra Aafer for kindly sharing the DroidAPIMiner source code with us. We also wish to thank Yanick Fratantonio for his comments on an early draft of the paper. This research was supported by the EPSRC under grant EP/N008448/1, by an EPSRC-funded "Future Leaders in Engineering and Physical Sciences" award, a Xerox University Affairs Committee grant, and by a small grant from GCHQ. Enrico Mariconti was supported by the EPSRC under grant 1490017, while Lucky Onwuzurike was funded by the Petroleum Technology Development Fund (PTDF).

References

[1] Google Play has hundreds of Android apps that contain malware. http://www.trustedreviews.com/news/malware-apps-downloaded-google-play, 2016.

[2] Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In *SecureComm*, 2013.

[3] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated Concolic Testing of Smartphone Apps. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, 2012.

[4] P. Andriotis, M. A. Sasse, and G. Stringhini. Permissions snapshots: Assessing users' adaptation to the android runtime permission model. In *IEEE Workshop on Information Forensics and Security (WIFS)*, 2016.

[5] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *Annual Symposium on Network and Distributed System Security (NDSS)*, 2014.

[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[7] S. Bernard, S. Adam, and L. Heutte. Using random forests for handwritten digit recognition. In *Ninth International Conference on Document Analysis and Recognition (ICDAR)*, 2007.

[8] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall. Brahmastra: Driving Apps to Test the Security of Third-Party Components. In *USENIX Security Symposium*, 2014.

[9] L. Breiman. Random forests. *Machine Learning*, 45, 2001.

[10] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: Behavior-based Malware Detection System for Android. In *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2011.

[11] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio. Detecting Android Malware Using Sequences of System Calls. In *Workshop on Software Development Lifecycle for Mobile*, 2015.

[12] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio. Acquiring and Analyzing App Metrics for Effective Mobile Malware Detection. In *IWSPA*, 2016.

[13] G. Canfora, F. Mercaldo, and C. A. Visaggio. An HMM and Structural Entropy based Detector for Android malware: An Empirical Study. *Computers & Security*, 61, 2016.

[14] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. MAST: Triage for Market-scale Mobile Malware Analysis. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2013.

[15] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu. StormDroid: A Streaminglized Machine Learning-Based System for Detecting Android Malware. In *AsiaCCS*, 2016.

[16] Y. Chen, M. Ghorbanzadeh, K. Ma, C. Clancy, and R. McGwier. A hidden Markov model detection of malicious Android applications at runtime. In *Wireless and Optical Communication Conference (WOCC)*, 2014.

[17] J. Clay. Continued Rise in Mobile Threats for 2016. http://blog.trendmicro.com/continued-rise-in-mobile-threats-for-2016/, 2016.

[18] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song. NetworkProfiler: Towards automatic fingerprinting of Android apps. In *IEEE INFOCOM*, 2013.

[19] W. Enck, P. Gilbert, S. Han, V. Tendulkar, 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. *ACM Trans. Comput. Syst.*, 32(2), 2014.

[20] W. Enck, M. Ongtang, and P. McDaniel. On Lightweight Mobile Phone Application Certification. In *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[21] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[22] E. Fix and J. Hodges. Discriminatory analysis, non-parametric discrimination. *USAF School of Aviation Medicine*, 31, 1951.

[23] J. Garcia, M. Hammad, B. Pedrood, A. Bagheri-Khaligh, and S. Malek. Obfuscation-resilient, efficient, and accurate detection and family identification of android malware. *Department of Computer Science, George Mason University, Tech. Rep*, 2015.

[24] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural Detection of Android Malware Using Embedded Call Graphs. In *ACM Workshop on Artificial Intelligence and Security (AISec)*, 2013.

[25] X. Ge, K. Taneja, T. Xie, and N. Tillmann. DyTa: Dynamic Symbolic Execution Guided with Static Verification Results. In *International Conference on Software Engineering (ICSE)*, 2011.

[26] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. *SIGPLAN Not.*, 40(6), 2005.

[27] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information Flow Analysis of Android Applications in DroidSafe. In *Annual Symposium on Network and Distributed System Security (NDSS)*, 2015.

[28] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[29] M. Hearst, S. Dumais, E. Osman, J. Platt, and B. Scholkopf. Support Vector Machines. *IEEE Intelligent Systems and their applications*, 13, 1998.

[30] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3), 1998.

[31] Y. Z. X. Jiang. Detecting passive content leaks and pollution in android applications. In *Annual Symposium on Network and Distributed System Security (NDSS)*, 2013.

[32] I. Jolliffe. *Principal Component Analysis*. John Wiley & Sons, Ltd, 2002.

[33] M. J. Kearns. *The computational complexity of machine learning*. MIT press, 1990.

[34] J. Kim, Y. Yoon, K. Yi, J. Shin, and S. Center. ScanDal: Static analyzer for detecting privacy leaks in android applications. In *MoST*, 2012.

[35] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer. Android Taint Flow Analysis for App Sets. In *SOAP*, 2014.

[36] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X.-y. Zhou, and X. Wang. Effective and Efficient Malware Detection at the End Host. In *USENIX security symposium*, 2009.

[37] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[38] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An Input Generation System for Android Apps. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2013.

[39] J. R. Norris. *Markov chains*. Cambridge University Press, 1998.

[40] J. Oberheide and C. Miller. Dissecting the Android Bouncer. In *SummerCon*, 2012.

[41] D. Octeau, S. Jha, and P. McDaniel. Retargeting Android Applications to Java Bytecode. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, 2012.

[42] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Annual Symposium on Network and Distributed System Security (NDSS)*, 2014.

[43] I. Polakis, M. Diamantaris, T. Petsas, F. Maggi, and S. Ioannidis. Powerslave: Analyzing the Energy Consumption of Mobile Antivirus Software. In *DIMVA*, 2015.

[44] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid Android: Versatile Protection for Smartphones. In *Annual Computer Security Applications Conference (ACSAC)*, 2010.

[45] V. Rastogi, Y. Chen, and X. Jiang. DroidChameleon: Evaluating Android Anti-malware Against Transformation Attacks. In *AsiaCCS*, 2013.

[46] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Android Permissions: A Perspective Combining Risks and Benefits. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2012.

[47] P. Schulz. Code protection in android. *Insititute of Computer Science, Rheinische Friedrich-Wilhelms-Universitgt Bonn, Germany*, 110, 2012.

[48] M. K. Shankarapani, S. Ramamoorthy, R. S. Movva, and S. Mukkamala. Malware detection using assembly and API call sequences. *Journal in Computer Virology*, 7(2), 2011.

[49] Statista. Global mobile OS market share in sales to end users from 1st quarter 2009 to 1st quarter 2016. http://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/, 2016.

[50] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *Annual Symposium on Network and Distributed System Security (NDSS)*, 2015.

[51] The Register. Google AdSense abused to distribute Android spyware. http://www.theregister.co.uk/2016/08/15/android_trojan_abuses_google_adsense/, 2016.

[52] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java Bytecode Optimization Framework. In *Conference of the Centre for Advanced Studies on Collaborative Research*, 1999.

[53] D. Venkatesan. Android.Bankosy: All ears on voice call-based 2FA. http://www.symantec.com/connect/blogs/androidbankosy-all-ears-voice-call-based-2fa, 2016.

[54] T. Vidas and N. Christin. Evading android runtime analysis via sandbox detection. In *AsiaCCS*, 2014.

[55] N. Viennot, E. Garcia, and J. Nieh. A measurement study of google play. 42(1), 2014.

[56] M. Y. Wong and D. Lie. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware. In *Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

[57] D. J. Wu, C. H. Mao, T. E. Wei, H. M. Lee, and K. P. Wu. DroidMat: Android Malware Detection through Manifest and API Calls Tracing. In *Asia JCIS*, 2012.

[58] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu. Effective Real-Time Android Application Auditing. In *IEEE Symposium on Security and Privacy*, 2015.

[59] L. K. Yan and H. Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *USENIX Security Symposium*, 2012.

[60] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in Android applications. In *ESORICS*, 2014.

[61] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. AppContext: Differentiating Malicious and Benign Mobile App Behaviors Using Context. In *International Conference on Software Engineering (ICSE)*, 2015.

[62] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[63] H. Ye, S. Cheng, L. Zhang, and F. Jiang. DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag. In *International Conference on Advances in Mobile Computing and Multimedia (MoMM)*, 2013.

[64] H. Zhang, Y. Cole, L. Ge, S. Wei, W. Yu, C. Lu, G. Chen, D. Shen, E. Blasch, and K. D. Pham. ScanMe Mobile: A Cloud-based Android Malware Analysis Service. *SIGAPP Appl. Comput. Rev.*, 16(1), 2016.

[65] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang. Leave me alone: App-level protection against runtime information gathering on Android. In *IEEE Symposium on Security and Privacy*, 2015.

[66] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *IEEE Symposium on Security and Privacy*, 2012.

[67] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Annual Symposium on Network and Distributed System Security (NDSS)*, 2012.