DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android

Yousra Aafer, Wenliang Du, and Heng Yin

Dept. of Electrical Engineering & Computer Science
Syracuse University, New York, USA
{yaafer, wedu, heyin}@syr.edu

Abstract. The increasing popularity of Android apps makes them the target of malware authors. To defend against this severe increase of Android malwares and help users make a better evaluation of apps at install time, several approaches have been proposed. However, most of these solutions suffer from some shortcomings; computationally expensive, not general or not robust enough. In this paper, we aim to mitigate Android malware installation through providing *robust* and *lightweight* classifiers. We have conducted a thorough analysis to extract relevant features to malware behavior captured at API level, and evaluated different classifiers using the generated feature set. Our results show that we are able to achieve an accuracy as high as 99% and a false positive rate as low as 2.2% using KNN classifier.

Keywords: Android, malware, static detection, classification

1 Introduction

As Android mobile devices are becoming increasingly popular, they are becoming a target of malware authors. To protect mobile users from the severe threats of Android malwares, different solutions have been proposed. Several systems have been proposed based on Android permission system. In [12], if an app requests a specific or a combination of critical permissions, a risk signal will be raised. In [22], several risk signals have been proposed depending on an app's requested permissions, its category, as well as the requested permissions from apps belonging to the same category. In [17], different risk scoring schemes have been designed using probabilistic generative models. However, the permission-based warning mechanisms fall short for several reasons:

- The existence of a certain permission in the app manifest file does not necessarily mean that it is actually used within the code. According to [13, 14, 26], a large percentage of Android apps are over-privileged.
- A large number of requested permissions, specially the critical ones, are actually not used within the application's code itself, but rather are required by the advertisement packages.
- Malware can perform malicious behavior without any permission [15].

Another direction to detect malicious activities in Android apps relies on the semantic information within the application bytecode. CHEX [16] statically

vets Android apps for component hijacking vulnerabilities through performing data flow analysis and conducting reachability tests on the generated system dependency graphs to detect potential hijack enabling flows. Similarly, Woodpecker [15] exposes capability leaks through using data flow analysis and exploring the reachability of a dangerous permission from a non-protected interface. While these approaches are effective in detecting the particular vulnerabilities that they target, they cannot be generalized to detect other malicious activities. DroidRanger [29], on the other hand, combines permission-based behavioral footprints and a heuristic based filtering scheme to detect malicious apps.

In this paper, we aim to overcome the shortcomings of the permission-based warning mechanisms and build a robust and lightweight classifier for Android apps that could be used for malware detection. To select the best features that distinguish between malware from benign apps, we rely on API level information within the bytecode since it conveys substantial semantics about the apps behavior. More specifically, we focus on critical API calls, their package level information, as well as their parameters.

Instead of following a heuristic based approach for identifying critical features for malware functioning, we have analyzed a large corpus of benign and malware samples, generated the set of APIs used within each app, and conducted a frequency analysis to list out the ones which are more frequent in the malware than in the benign set. Furthermore, for certain critical APIs which were frequent in both sample sets, we have conducted a simple data flow analysis on the malware APK samples to identify potentially dangerous inputs. We generated a list of frequently used parameters, thoroughly examined them to filter out the dangerous ones and flagged all apps that request them. To perform API level feature extraction and data flow analysis, we have developed a tool called DroidAPIMiner built upon Androguard [2] reverse engineering tool. We use RapidMiner [7] to build the classification models.

In summary, the contributions of this paper are as follows:

- We introduce a robust and efficient approach for describing Android malware that relies on the API, package, and parameter level information.
- Based on the identified feature set of Android malware, we provide valuable insights about malware behavior at API-level.
- We produce and evaluate different classifiers for Android apps. Our testing shows that some of them achieve a high accuracy and low false positive rate compared to the permission-based classifiers. In fact, KNN achieves a 99% accuracy and 2.2% false positive rate.

2 Approach Overview

In our work, we follow a generic data mining approach that aims to build a classifier for Android apps. The classifier should be able to automatically learn to identify complex malware patterns and make smart decisions based on that. The classifier should also be able to generalize from the input set to correctly predict an accurate class of given new apps. As depicted in Fig. 1, our approach

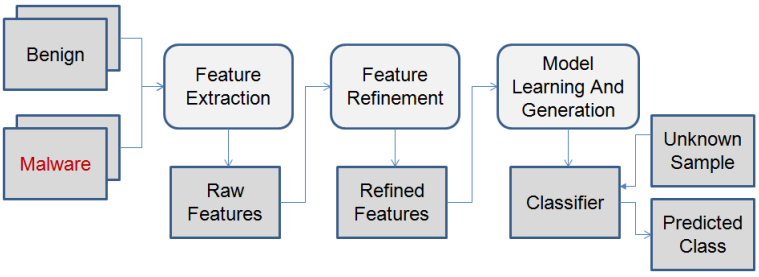


Fig. 1. Our Approach

is divided into three phases: feature extraction, feature refinement, and models learning and generation.

During the feature extraction phase, we statically examine the collected benign and malware APK samples to determine and extract the necessary features for malware to function. In selecting the feature set, we focus on some semantic information encapsulated within the bytecode of apps. More specifically, we extract API calls and their package level information. Besides, we extract the requested permissions of the apps for the generation of the baseline model.

During the feature refinement phase, we remove the API calls that are exclusively invoked by third-party packages such as advertisement packages. We reduce our feature set further to include only those APIs whose support in the malware set is significantly higher than in the benign set. For those APIs which were frequent in the two sets, we perform data flow analysis to recover their parameter values and select only the APIs that invoke dangerous values. Subsequently, for each APK file, we generate a set of feature vectors along with associated class labels, i.e. malware or benign. For the last two steps, we have implemented DroidAPIMiner, a python program that import libraries from Androguard static analysis tool for Android apps [2]. Section 3 will be dedicated to discuss in more details how we conduct feature extraction and refinement. We discuss in Section 4 some of the insights that we have gained based on the identified features.

During the model learning and generation phase, we feed the representative vectors to standard classification algorithms that build the models by learning from them. We have generated 4 different classifiers: ID5 DT [20], C4.5 DT [21], KNN [8] and SVM [25]. We test the generated classifiers to estimate the accuracy using split validation. Two thirds of the data set are randomly selected for training and the rest one third is dedicated for testing. For this step, we use RapidMiner [7] to generate the classification models and evaluate them. In Section 5, we perform the classification and evaluate the models.

3 Feature Extraction and Refinement

In this section, we aim to systematically determine and extract necessary features for malware functioning. Android app's bytecode contains information that

could be used to describe its behavior. From the bytecode, we can retrieve information ranging from coarse-grained levels as packages to fine-grained levels as instructions. We do not perform sophisticated program analysis because it is computationally expensive. Rather, we focus on extracting package and API level information since they clearly capture the app's behavior. More specifically, we consider class name, method name, and some parameters of the callee and the package name of the caller, which we will describe in the next subsections.

3.1 Extraction of Dangerous APIs

Contrary to previous work, we do not follow a heuristic-based approach to identify dangerous APIs for malware functioning. Instead, we aim to reliably identify the major APIs that malwares invoke by statically analyzing our samples.

Effectively, we have statically analyzed a large set of malware and benign apps and generated a list of distinct API calls within each set. A distinct API refers to a distinct combination of Class Name, Method Name, and Descriptor. We then conduct a frequency analysis to select those APIs which are more used in the malware than in the benign set. We further refine the API list to include only those with a usage difference higher or equal to a certain threshold.

3.2 Extraction of Package Level Information

Most of Android apps contain one or more third-party packages (according to our analysis, 71 % of the benign apps contain at least one advertisement package). These packages often exhibit some suspicious behavior. For instance, many ads use encryption to hinder their removal. Also, `getCellLocation()` and `getDeviceId()` methods are often called by ad kits for users' identification and tracking purposes. We aim to identify at what package level a certain API is invoked. To achieve this goal, we have performed the following tasks:

- **Extract advertisement and similar packages:** Using Androguard, we generate all distinct packages invoked within each APK in our collected sample. We remove from the generated packages names all common packages such as Android specific packages, Java packages, etc. We inspect the remaining items and compile a list of advertisement, web tracking, web analysis and application ranking packages. In total, we have identified around 412 distinct advertisement and similar packages. Some commonly used advertisement packages are: Admob, Flurry, Millennialmedia, Inmobi, Adwhirl, Adfonic, Adcenix, etc.
- **Identify calling packages:** We check at what package a certain API is called. In other words, we distinguish if an API is invoked only by a third-party package, only by the application specific packages, or by both. We white-list any APIs that are exclusively invoked by third-party packages.

3.3 Extraction of APIs Parameters

Certain frequent APIs in the malware set did not yield to a high support difference between the malware and the benign sample as they were also common in the benign sample. For example, some methods within string manipulation and IO classes are almost as frequent in the malicious set as in the benign set. To

increase this difference, we have performed data flow analysis on these specific APIs in order to recover the parameters values that have been passed to them through inspecting the registers invoked.

Table 1. Categorization of Parameters to Frequently Used Malware APIs

Classes	Methods	Parameter Category
Intent IntentFilters	setFlags, addFlags, setDataAndType, putExtra, init	Flag is either: CALL, CONNECTIVITY, SEND, SENDTO, or BLUETOOTH
ContentResolver	query, insert, update..	URI is either: Content://sms-mms, Content://telephony, Content://calendar, Content://browser/bookmarks, Content://calllog, Content://mail, or Content://downlaods
DataInputStream BufferedReader DataOutputStream DataOutputStream	init, writeBytes...	Reads from process Reads from connection Uses SU command
InetSocketAddress	init	parameter IP is explicit or port is 80
File Stream StringBuilder String StringBuffer	init, write, append, indexOf, Substring	Dangerous Command such as: su, ls, loadjar, grep, /sh, /bin, pm install, /dev/net, insmod, rm, mount, root, /system, stdout, reboot, killall, chmod, stderr Accesses external storage or cache Contains either: An identifier (e.g. Imei), an executable file(e.g. .exe, .sh), a compressed file (e.g. jar, zip), a unicode string, an sql query, a reflection string, or a url

Based on our initial investigation, these APIs generated distinct parameters which resulted in a big number of features. To reduce the parameter feature set, we have categorized the parameters based on different criteria. Table 1 includes the APIs on which we have performed the data flow analysis along with the criteria that we have adopted to categorize their input parameters.

4 Insights in API-Level Malware Behavior

Based on the API level analysis, we have identified the top APIs that Android malwares invoke. Fig.2 shows the top 20 APIs that produce the highest difference of usage between malware and benign apps. As illustrated, we get a better difference after filtering out third-party packages. For example, the method `init` in `Java.Util.TimerTask` initially produced 14% usage difference between the two sets. This difference increased to 28% after whitelisting this API in third-party packages since it is mainly invoked by them in the benign sample.

We discuss here some of the top commonly used malware features that our study generated after refining the initial feature set. To help understand malware behavior and gain more insight into what resources are accessed and what actions are performed, we classify the APIs by the type of requested resources and utilities. At the end of the section, we present the data flow analysis results.

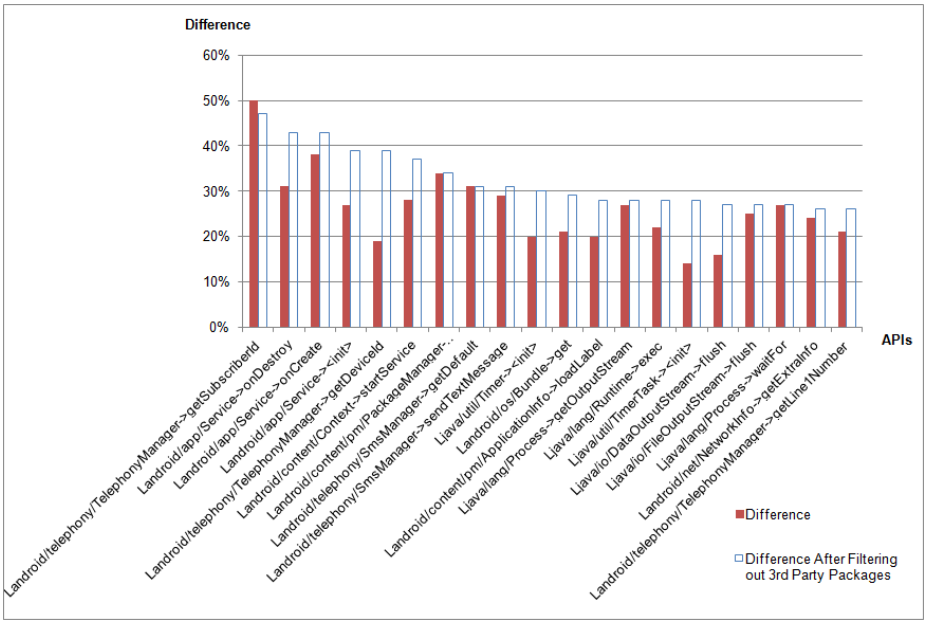


Fig. 2. Top 20 APIs with the Highest Difference Between Malware and Benign Apps

4.1 Application-specific resources APIs

Content Resolver: This class provides access to content providers. It processes requests (CRUD operations) by directing them to the appropriate content provider. The most frequent methods used in this class by malware are `insert()`, `delete()` and `query()`. This latter can be invoked to grab sensitive information from content providers of other apps if they are not protected by permissions. As stated in [5], some vendor pre-installed apps have implicitly exported content providers which allowed other apps to successfully obtain sensitive information from them without acquiring the necessary permissions.

Context: Context class provides global application information such as its specific assets, classes, and resources. `startService()` is very frequently used methods within this class with a support of more than 70% in malware and less than 34% in benign ones. This API can be invoked to start a given service in the background without interacting with the user. `getFilesDir()` and `openFileOuput()` are other frequent APIs in this class that malwares call to create files and get their absolute paths. `getApplicationInfo()` is often used by malwares for obtaining various information about the app such as whether it's debuggable, installed on external storage, holds factory test flag, etc.

Intents: Intents allow launching other activities and services and interacting with the phone's hardware. The most frequent APIs used by malwares within Intents are `setDataAndType()`, `setFlags()` and `addFlags()`. `setDataAndType()` allows setting the URI path for the intent data with an explicit MIME data

type. As stated in the official documentation of Android [4], this method should "very rarely be used" since it allows to override the ordinary inferred MIME type of data of a newly specified MIME type. `setFlags()` and `addFlags()` are used to set the old flags or add new ones to the intent to specify how it should be handled. Depending on the parameter flag to these APIs, malware controls the associated component such as running it with foreground priority.

4.2 Android framework resources APIs

ActivityManager: This class allows interacting with other activities running in the system. The method `getRunningServices()` is often invoked by malware to inquire whether a certain service (like Anti-virus) is currently executing. `getMemoryInfo()` is also frequently invoked by malware and might be used to check how close the system to have no enough memory for other background process and thus needing to start killing other processes. `restartPackage()` is often invoked by malware to kill other apps' services. According to Android's documentation [1], the original behavior of this method is no longer available to apps as it "allows them to break other applications by removing their alarms, stopping their services, etc".

PackageManager: This class contains information about the application packages installed on the device. Malicious apps call `getInstalledPackages()` to scan the system against a list of known anti-virus and take an appropriate action based on that (e.g. remain dormant, kill the anti-virus process, etc.) .

Telephony/ SmsManager and telephony/ gsm/ SmsManager: These classes allows managing various SMS operations. Malware authors invoke many methods within theses classes. `sendTextMessage()` is very frequently used by malwares authors to send sms messages to premium rate numbers without the user's consent and thus incur financial losses. Examples of SMS Trojans include malware belonging to the following families: SpyEye, OpFake, Gemini, etc.

TelephonyManager: This class retrieves various information about telephony services on the device. The most frequently used APIs by malwares are: `getSubscriberId()`, `getDeviceId()`, `getLine1Number()`, `getSimSerialNumber()`, `getNetworkOperator()`, and `getCellLocation()`. Malware authors collect these private data and send it to remote servers to build users profiles and track them. As illustrated in Fig. 2, `getSubsriberId()` is the mostly used API by our malware sample.

4.3 DVM related resources APIs

DexClassLoader: This class allows loading classes from external .jar and .apk files containing a classes.dex. `loadClass()` is one of the most frequently invoked APIs by malware and is used to execute code not installed as part of the app and consequently evade malware detection techniques that rely on static analysis.

Runtime and System: Runtime class allows apps to interact with the environment in which they are running. Malware invokes `Runtime.getRuntime.exec()` method to execute dangerous Linux commands along with the supplied arguments in a newly spawned native process and thus avoid the normal execution

lifecycle of the program. System class provides system related facilities such as standard input, output and error output streams. `loadLibrary()`dynamically loads native libraries and can be used maliciously through running native code exploiting some known system vulnerabilities.

4.4 System resources APIs

ConnectivityManager, NetworkInfo, and WifiManage: These classes provide network related functionalities such as answering queries about different connections (Wifi, GPRS, UMTS) and network interfaces. Android malware calls APIs within ConnectivityManager class (`getNetworkInfo()`), NetworkInfo (`getExtraInfo()`, `getTypeName()`, `isConnected()`, `getState()`), and within WifiManager(`setWifiEnabled()` and `getWifiState()`) to establish a network connection and interact with malicious remote servers.

HttpURLConnection and Sockets: APIs within these classes are used to send and receive data over the web and establish communication with remote servers. The most frequent APIs used by malwares in HttpURLConnection are `setRequestMethod()`, `getInputStream()`, and `getOutputStream()` which manage transferring data between the malware apps and the malicious servers. Similarly, malware applications often invoke `getInputStream()` and `getOutputStream()` in Socket class for the same purpose. We have also noted a heavy use of InetSocketAddress which implements an IP socket address given an IP address and a port number.

OS package: A lot of frequently used APIs in malware belong to OS package which allows message passing, ipc services, process and threads management. `sendMessage()` method in os.Handler class inserts messages into message queues of different executing threads, while `obtainMessage()` retrieves messages from the message queues. Malware authors often invoke `myPid()` and `killProcess()` in Process class to request killing processes based on a given pid. However, the kernel will impose restrictions on which processes an application can actually kill [6]; only apps and packages sharing common UIDs can actually kill each other. Unfortunately, these restrictions will not prevent Android malware from killing processes beyond their scope once they can root the device.

IO Package: IO package provides IO processing services such as reading and writing to streams, files, internal memory buffers, etc. Malwares invoke APIs within IO.DataOutputStream (such as `writeBytes()`) to write data and upload files through a URL connection. Similarly, they call APIs in IO.DataInputStream (such as `readLines()`, `available()`) to read and download malicious payloads from a certain URL connection. Methods within IO.FileOutputStream (such as `write()`) are used to write the malicious content downloaded from a remote server to local files. `mkdir()`, `delete()`, `exists()` and `ListFiles()` are other used APIs in IO.File by malware for file management.

4.5 Utilities APIs

String, StringBuilder and StringBuffer: These classes provide an interface for creating and manipulating strings. Malware heavily call `substring()`,

`indexOf()`, `getBytes()`, `valueOf()`, `replaceAll()`, and `Append()`. These methods can be used for code obfuscation, construction of payloads to be sent to servers, and evasion of static malware detection techniques through dynamically creating URLs, parameters to reflection APIs, and dangerous Linux commands.

Timer: Timers facilitate scheduling one-shot or recurring tasks for future execution. Malware can invoke APIs within this class (such as `schedule()` and `cancel()`) to avoid dynamic analysis by remaining dormant until a fixed date is reached, or until a specific event has been fired.

ZipInputStream: This class allows decompressing data from an InputStream ZIP archive. Malwares rely on methods in this class to decompress and read data from compressed files (.jar, .apk, .zip) downloaded during execution or originally attached to the app. Commonly used APIs by malware in this class are `read()`, `close()`, `getNextEntry()` and `closeEntry()`.

Crypto: This package serves as an interface for implementing cryptographic operations such as encryption, decryption, and key agreement. Methods within Crypto.Cipher such as `getInstance()` and `doFinal()` transform a given input to an encrypted or decrypted format while `Crypto.spec.DESKeySpec()` allows specifying a DES key. These methods can be used for code obfuscation and avoiding static detection through encrypting root exploits, SMS payloads, targeted premium SMS numbers, and URLs to remote malicious servers.

w3c.dom: This package provides the official w3c Java interfaces for the Document Object Model (DOM), which is used in apps for XML document processing. Malwares use several APIs in w3c.dom such as `getDocumentElement()`, `getElementByTagName()`, and `getAttribute()` to parse XML files. XML can be used by malwares to establish bot communication, encode data, and process local configuration files.

4.6 Parameters Features:

Based on the data flow analysis that we have conducted, we obtained the frequent parameters (categorized as discussed in Table 1) that are used by malwares applications more often than the benign ones in certain API invocations. Table 2 depicts some of the top invoked parameters types that yield to the highest support difference between the malware and benign sample.

From the data flow analysis results depicted in Table 2, we can gain more insight on Android malware behavior. A large percentage of String manipulation operations are performed on dangerous Linux commands (such as SU, mount, sh, bin, pm install, killall, chmod). These commands are mainly used by malware authors to root the phone and exploit some well known vulnerabilities. After getting superuser privilege, malwares perform various dangerous Linux operations through invoking runtime.exec(). Most of the ContentResolver operations are performed on SMS, MMS, telephony or call log content providers.

Table 2. Some Frequent API Parameters in Malware

Class	Method	Parameter type	Difference (%)
StringBuilder	append	Dangerous command	35.95
ContentResolver	query	SMS or MMS	23.65
StringBuilder	append	Unicode string	23.6
StringBuilder	init	Dangerous command	23.07
DataOutputSream	writebytes	Reads from process	21.80
DataOutputSream	init	Reads from process	21.62
runTime	exec	Dangerous command	21.27
InetSocketAddress	init	Port 80	19.91
StringBuilder	append	Compressed file	19.58
DataInputStream	init	Reads from connection	19.27
String	valueOf	Unicode string	18.05
StringBuilder	append	File manupilation	17.79
File	init	Accesses external storage	16.92
InetSocketAddress	init	Explicit IP	14.87
String	getBytes	URL manupilation	14.05
Intent	setFlags	SendTo	12.94
Intent	setFlags	Call	11.67
ContentResolver	query	Telephony	10.88
Intent	setFlags	Send	10.47
ContentResolver	query	Call_log	10.12

5 Classification and Evaluation

5.1 Data Set

To extract malware and benign apps' features, generate and evaluate the classification models, we have collected and analyzed around 20,000 apps. Our malware sample consists of 3987 malware apps that we collected from different sources (McAfee and Android Malware Genome Project [3]). The malware sample belongs to different Android malware families. Our benign sample consists of the top 500 free apps in each category in Google Play (around 16000 apps) that we collected in July 2012.

5.2 Classification Models

As discussed earlier, our objective is to build a model that classifies unknown apps as either benign or malware. For that, we have employed four different algorithms for the classification: ID3 DT [20], C4.5 DT [20], KNN [8], and linear SVM [25]. These inducers belong to different family of classifiers. C4.5 and ID3 are related to decision trees and KNN belong to Lazy classifiers. SVM is a supervised learning method that proceeds through dividing the training data by an optimal separating hyperplane. We have decided to employ algorithms from different classifiers because we hope that they will produce different classification models for Android apps. Our analysis shows that KNN and ID3 DT models lead to a better accuracy compared to the other models.

To test our generated classification models, we use split validation. That is, we randomly split our dataset into training (2/3) and testing set (1/3). We build the classification models based on the training set and feed the testing instances to evaluate the models. To evaluate each classifier's performance, we measured the True Positive Ratio (TPR), i.e., the proportion of malware instances that were correctly classified:

$$TPR = \frac{TP}{TP + FN}$$

where TP is the number of malware apps correctly identified and FN is the number of malware apps classified as benign apps. Similarly, we measure the True Negative Ratio (TNR), i.e., the proportion of benign instances that were correctly classified:

$$TNR = \frac{TN}{TN + FP}$$

where TN is the number of benign apps correctly identified and FP is the number of benign apps identified as malware apps. To capture the overall performance, we measure the models' accuracy, i.e., the total number of benign and malware instances correctly classified divided by the total number of the dataset instances:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

By means of our collected dataset, we conducted different experiments to find the optimum feature set that will produce the best cut between the malware and benign sample.

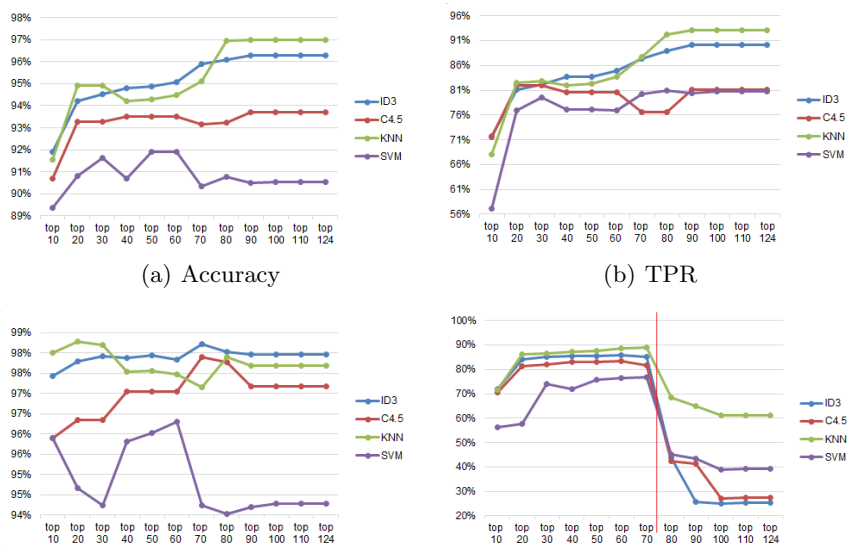
5.3 Permission-Based Feature Set

In the first experiment, we extract the permissions requested by malware and benign apps and obtain their perspective percentage usage in the two sets. We then rank the permissions based on the difference usage and took the top k permissions that are more frequently requested in malware than in benign apps. To determine the optimum k permissions, we evaluate the performance of the models for k = 10, 20, 30..., up to 124.

Fig. 3 depicts the results obtained for the permission-based feature set in terms of accuracy, TPR, and TNR. As illustrated, the models' accuracy increases as the feature set includes more permissions. It should be noted that only 64 permissions were more frequent in the malware set than in the benign set, which means that after the top 64 permissions, the classifiers start to learn also from the permissions that are frequent in the benign set. This makes the classifiers not solid enough since they can fail to detect malicious apps in the following two scenarios. First, malware authors can easily defeat the permission-based classifiers through merely declaring "benign" permissions in the manifest file. Second, the classifiers will not be able to correctly classify repackaged android malware; which is based on legitimate apps but embeds extra payload to achieve a malicious goal. The manifests of the repackaged apps include both the original permissions of the benign app and the permissions needed for the malicious

behavior and thus confuse the classifiers.

To demonstrate that the permission model is not robust enough, we designed an experiment in which we modify our malware set and feed it to the classifiers. In each malware manifest, we declare 10 new permissions (the top 10 in the benign set) and keep everything else unchanged. As shown in Fig. 3(d), when the feature set contains the permissions used in the benign set, the classifiers are not able to correctly classify the malware set. In fact, using the top 80 permissions, the classification rate of KNN drops to 67% and of ID3 to 43%.



(a) Accuracy

(b) TPR

(c) TNR

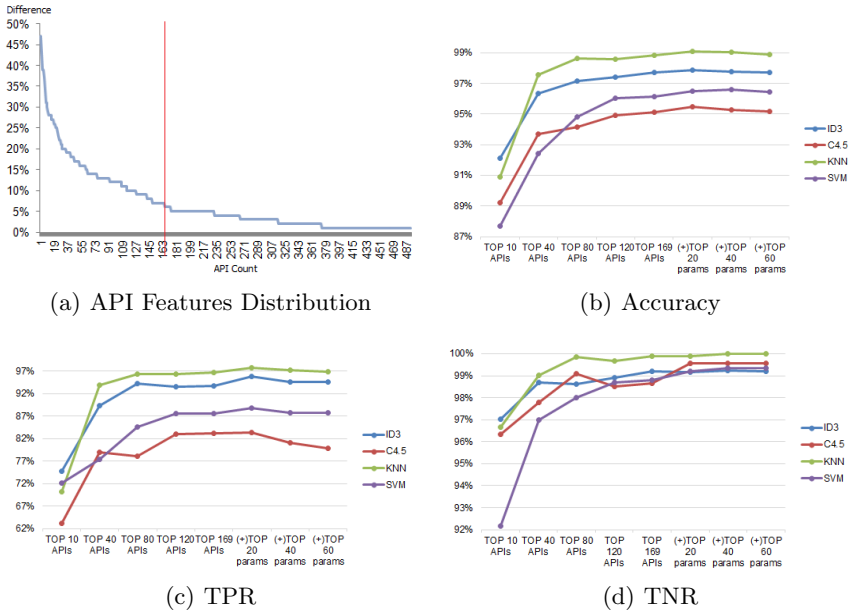
(d) Classification Rate of Modified Malware Set

Fig. 3. Performance of Permission-based Models

5.4 API-Based Feature Set with Package Level and Parameter Information

In the second experiment, our feature vector includes the generated APIs within each set, which make up in total 8375 distinct APIs. We also embed package level information. That is, we white-list the APIs that are exclusively called by third-party packages. We specifically filter out these APIs to avoid the case where a benign app might be classified as malicious if a third-party package invokes a possibly "malicious" API. Consequently, the support of white-listed APIs drops in the benign set.

We conduct a frequency analysis and took only the APIs whose usage in the malware set is higher than in the benign set. Based on this, we have reduced our features to 491 APIs. As shown in Fig. 4(a), a large portion of these APIs have



(a) API Features Distribution

(b) Accuracy

(c) TPR

(d) TNR

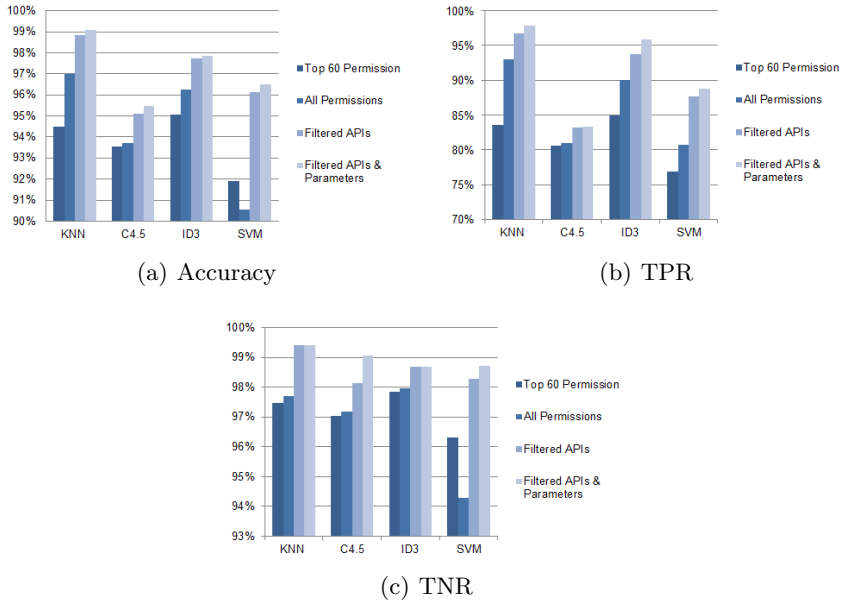
Fig. 4. Performance of API-based Models

a usage difference of less than 6% which will result in creating more noise in the classifiers and slow down the learning process. To solve this issue, we further refine our feature set to include only the top 169 APIs (with a usage difference greater or equal to 6%).

We generate the classification models for the top k (10, 40, 80, 120 and 169) API features and evaluate their performance. As depicted in Fig. 4, using the top 169 API based features, we achieve the highest accuracy, TPR and TNR using KNN. C4.5 is the worst performing model as it barely achieves 83% TPR.

In the same experiment, we also include the parameter-based features obtained using data flow analysis on the original set. We re-generate the models and evaluate them after adding 20, 40, and 60 parameters to the 169 filtered APIs. As shown in Fig. 4, by adding the top 20 used parameters, we are able to achieve the highest accuracy (99%) and TPR (97.8%) using KNN algorithm. The other algorithms also perform better with the newly added parameter-based feature set.

Unlike permission-based classifiers, it is not possible to trick API-based classifiers through declaring benign APIs, because the models do not rely on benign features to classify a given app. Rather, they only rely on the APIs (along with parameters) that are more frequently used in malware than in benign apps.



(a) Accuracy

(b) TPR

(c) TNR

Fig. 5. Models Comparison

5.5 Models Comparison

To show the improvement achieved over the experiments performed, we plot the accuracy, TPR, and TNR of the classification models together as depicted in Fig. 5. We consider two permission models. The first one is trained on the top 60 frequent permissions in malware and the second one on all the permissions. For the API filtered model, the feature vector includes all the top 169 features. The last model that we consider is trained on the top 169 filtered APIs along with the top 20 frequent parameters in certain APIs within malware.

As shown in Fig. 5, our API based features performs better than the permission-based one. We were able to improve the accuracy, TPR and TNR of the models by embedding package and some parameter features to our original features. KNN is the best performing model, followed by ID3, SVM then C4.5.

5.6 Processing Time

It is evident that the processing time is a crucial metric for a scalable detection system. In this section, we report the execution time of DroidAPIMiner which consists of the time required to de-assemble an apk file and to extract the API and parameter feature set. We also report the time that RapidMiner requires for applying different classification models to classify a new instance. We perform the analysis an Intel Core i5-2430M machine with 6GB of memory.

Fig. 6 shows the distribution of DroidAPIMiner processing time among the collected apps sample. As depicted in the graph, more than 80% of the apps

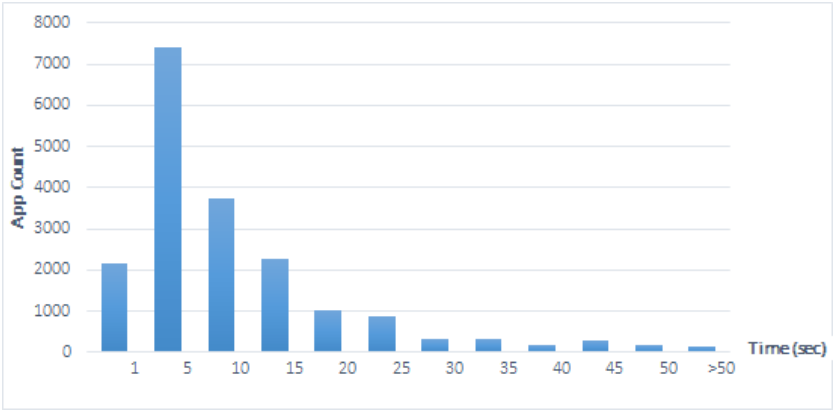


Fig. 6. Distribution of DroidAPIMiner Processing time

require less than 15 sec to be analyzed by DroidAPIMiner. Besides, as shown in Table. 3 applying KNN algorithm to classify new inputs is quite fast and takes less than 10 sec. In total, our detection system requires on average about 25 sec to classify an apk file as either benign or malicious, which makes it efficient enough to be deployed on either mobile devices and back-end servers.

Table 3. Processing Overhead of the Classification Algorithms

Algorithm	Model Application and Classification time (sec)
ID3	185.0 +- 32.0
KNN	9.0 +- 1.0
C4.5	21.0 +- 4.0
SVM	160.2 +- 40.0

6 Discussion

In this section, we discuss some potential evasion techniques that malware authors may adopt in order to thwart our classifiers. Furthermore, we discuss how our tool handles these cases.

– Reflection: Malware authors may use reflection to easily obfuscate any dangerous API call and thus evade the static detection of the occurrence of that API by our analysis tool. However, it should be noted that our study has shown that reflection APIs are more frequently used by our malware set than in the benign set, which makes them part of the feature vector for the classification.

– Native Code: To avoid static detectors at the bytecode level, malwares sometimes embed malicious payload within native content. Since our detection tool

only works at bytecode level, it will not be able to detect any dangerous methods invoked. However, the use of JNI calls such as `System.loadLibrary()` is also used as a classification feature by our tool.

- Bytecode Encryption: To prevent reverse engineering of Java code, malware authors may encrypt their code and allow the decryption at runtime. Our tool considers decryption APIs as a classification feature.
- Dynamic Loading: As discussed earlier, DexClassLoader allows loading classes from .jar and .apk files at runtime and executing code not installed as part of an app. `loadClass()` in DexClassLoader also belongs to our feature set.
- More Benign Calls: Since our classifiers rely on the frequency of API calls, malware authors might think of introducing more benign API calls into their code. However, our tool is not susceptible to this problem, because we do not rely on the occurrence of benign API calls as a feature for the classification. Rather, we only consider the occurrence of malicious call as a feature.

7 Related Work

Several studies have been conducted in the field of Android malware detection. One much-studied direction focuses on the permission system. Kirin [12] blocks apps that declare risky permission combinations or contain any suspicious action strings used by activities, services or broadcast receivers. Zhou et al. [29] detect Android malware based on the similarities of the requested permissions and the behavioral footprints to different known malware families. Sarma et al. [22] propose different risk signals based on the requested permissions, category as well as requested permissions of apps belonging to the same category. In another work, Sarma et al. [17] employ probabilistic generative models to compute a real risk score of Android apps based on the permissions that they request.

Another direction of related work relies on system level events to detect possible malicious behavior. Schmidt et al. [23] extract library and system function calls from Android executables and compare them to malware executables to classify apps. Crowdroid [10] collects system call traces of running apps on different Android devices and applies clustering algorithms to detect malwares.

More similar research to our study rely on semantics within the bytecode to detect specific vulnerabiltities in Android applications. Potharaju et al. [19] aim to detect plagiarized apps through different detection schemes relying on symbol tables and method-level Abstract Syntactic Tree fingerprints. In [28], Zhou et al. aim to systematically detect and analyze repackaged apps on third party Android markets based on fuzzy hashing techniques.

Other related work for detecting malware through bytecode level information have been proposed by Blasing et al. [9] and Zhou et al. [29]. However, the first one (AASandbox) relies on a trial and error approach to identify suspicious patterns in the source code, while DroidRanger performs the detection with regards to a heuristic based filtering. In our work, we conduct a thorough frequency analysis of API calls within benign and malware apps to extract malware features and employ machine learning to get the most relevant ones.

A different direction for detecting Android malware relies on dynamic anal-

ysis. Andromaly [24] continuously monitors various system metrics to detect suspicious activities through applying supervised anomaly detection techniques. In [11], Enck et al. perform dynamic taint analysis to track the flow of private and sensitive data through third party apps, and detect any leakage to remote servers. Portokalidis et al. [18] propose a security model for protecting mobile devices which performs multiple attack detection techniques simultaneously on remote servers hosting an exact replica of the devices. Lok and Yin [27] present DroidScope, a virtualization based platform for Android malware analysis. It rebuilds both the operating system and Java level semantics, and enables instrumentation of the Dalvik and native instructions. Consequently, Droidscope can be used to understand the behavior of malware both at the native code level as well as at the interaction with the system.

8 Conclusion and Future Work

We have presented a robust and lightweight approach for detecting Android malware based on different classifiers. To predict whether an app is benign or malicious, the classifiers rely on the semantic information within the bytecode of the applications ranging from critical API calls, package level information and some dangerous parameters invoked. Rather than following a heuristic based approach for determining the feature vector of the classifiers, we have statically analyzed a large corpus of Android malwares belonging to different families and a large benign set belonging to different categories. We have conducted a frequency analysis to capture the most relevant API calls that malware invoke, and refined the feature set to exclude API calls made by third-party packages. We performed a simple data flow analysis to get dangerous input to some API calls.

Our classification results indicate that we are able to achieve a better accuracy, TPR and TNR using a combination of API, package, and parameter level information in comparison to the permissions-based feature set. As future work, we plan to further reduce the false positives and negatives through analyzing the samples that were not correctly classified and finding out the reasons behind the misclassification.

9 Acknowledgment

We would like to thank anonymous reviewers for their comments. This research was supported in part by NSF grants #1017771, #1018217, #1054605, and #1116932, Google research grant, and McAfee research grant. Any opinion, findings, conclusions, or recommendations are those of the authors and not necessarily of the funding agencies.

References

1. ActivityManager. `http://developer.android.com/reference/android/app/ActivityManager.html`.
2. Androguard. http://code.google.com/p/androguard/.
3. Android Malware Genome Project. `http://www.malgenomeproject.org/`.
4. Intent. `http://developer.android.com/reference/android/content/Intent.html`.

5. Malware that Takes Without Asking. http://labs.mwrinfosecurity.com/tools/2012/03/16/mercury/documentation/white-paper/malware-that-takes-without-asking/.
6. Process. http://developer.android.com/reference/android/os/Process.html.
7. RapidMiner. http://rapid-i.com/content/view/181/190/.
8. D. W. Aha, D. Kibler, and M. K. Albert. Instance-Based Learning Algorithms. *Machine Learning, 6:3766*, 1991.
9. T. Blasing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak. An Android Application Sandbox System for Suspicious Software Detection. *MALWARE*, 2010.
10. I. Burguera, U. Zurutuza, and S.Nadijm-Tehrani. Crowdroid: Behavior-Based Malware Detection System for Android. *SPSM*, 2011.
11. 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. *USENIX, OSDI*, 2011.
12. W. Enck, M. Ongtang, and P. McDaniel. On Lightweight Mobile Phone Application Certication. *CCS*, 2009.
13. A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystied. *CCS*, 2011.
14. A. P. Felt, K. Greenwood, and D. Wagner. The Effectiveness of Application Permissions. *USENIX, WebApps*, 2011.
15. M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. *NDSS*, 2012.
16. L. Lu, Z. Li, Z. Wu, W. Lee, and G.i Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. *CCS*, 2012.
17. H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, and R. Potharaju. Using Probabilistic Generative Models for Ranking Risks of Android Apps. *CCS*, 2012.
18. G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid Android: Versatile Protection for Smartphones. *ACSAC*, 2010.
19. R. Potharaju, A. Newell, C. Nita-Rotaru, , and X. Zhang. Plagiarizing Smartphone Applications: Attack Strategies and Defense. *ESSoS*, 2012.
20. J. R. Quinlan. Induction of Decision Tree. *Machine Learning, Vol. 1, No. 1. pp. 81-106.*, 1986.
21. J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.
22. B. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Android Permissions: A Perspective Combining Risks and Benets. *SACMAT*, 2012.
23. A.-D. Schmidt, R. Bye, H.-G. Schmidt, J.Clausen, O. Kiraz, K. A. Yuksel, S. A. Camtepe, and S. Albayrak. Static Analysis of Executables for Collaborative Malware Detection on Android. *ICC*, 2009.
24. A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss. Andromaly: a Behavioral Malware Detection Framework for Android Devices. *Journal of Intelligent Information Systems archive Volume 38 Issue 1*, 2012.
25. V. Vapnik. *The Nature of Statistical Learning Theory*. Springer-Verlag, NY, 1995.
26. X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Permission Evolution in the Android Ecosystem. *ACSAC*, 2012.
27. L. K. Yan and H. Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. *USENIX, Security*, 2012.
28. W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. *CODASPY*, 2012.
29. 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. *NDSS*, 2012.