StormDroid: A Streaminglized Machine Learning-Based System for Detecting Android Malware

Sen Chen
Dept. of Computer Science
East China Normal University
ecnuchensen@gmail.com

Minhui Xue
East China Normal University
NYU Shanghai
minhuixue@nyu.edu

Zhushou Tang
Shanghai Jiao Tong University
Pwnzen Infotech Inc.
ellison.tang@gmail.com

Lihua Xu
Dept. of Computer Science
East China Normal University
lhxu@cs.ecnu.edu.cn

Haojin Zhu
Dept. of Computer Science
Shanghai Jiao Tong University
zhu-hj@cs.sjtu.edu.cn

ABSTRACT

Mobile devices are especially vulnerable nowadays to malware attacks, thanks to the current trend of increased app downloads. Despite the significant security and privacy concerns it received, effective malware detection (MD) remains a significant challenge. This paper tackles this challenge by introducing a streaminglized machine learning-based MD framework, *StormDroid*: (i) The core of *StormDroid* is based on machine learning, enhanced with a novel combination of contributed features that we observed over a fairly large collection of data set; and (ii) we streaminglize the whole MD process to support large-scale analysis, yielding an efficient and scalable MD technique that observes app behaviors statically and dynamically. Evaluated on roughly 8,000 applications, our combination of contributed features improves MD accuracy by almost 10% compared with state-of-the-art antivirus systems; in parallel our streaminglized process, *StormDroid*, further improves efficiency rate by approximately three times than a single thread.

CCS Concepts

•Security and privacy → Malware and its mitigation; Mobile platform security; •Computing methodologies → Supervised learning by classification;

Keywords

Malware Detection; Machine Learning; StormDroid

1. INTRODUCTION

The number of mobile devices has been skyrocketing in the past several years, eclipsed traditional computers as points of entry to the Internet in an era of "always-on, always-connected" communications. Mobile devices are able to access data in a wide variety of forms, from text messaging to live applications for streaming services, gaming, and beyond. They are increasingly characterized by their pervasiveness and connectedness in all facets of daily life. A recent report[1] shows that there are about 1.6 million apps in the Google Play Store in July 2015.

Unfortunately, with recent trends in increased apps downloading, mobile devices are especially vulnerable to Android malware, often spread by masquerading as useful programs. For example, mobile device malware rates[2] – reflecting the number of devices attacked but not infected – surged 75% from 2013 to 2014. Worse yet, attackers have also innovated in the way they infect new devices, such as using third party developer stolen keys to sign malware samples, or taking advantage of zero days exploits to get root access to the device.

Recent Android's countermeasures aim to solve this problem via one of the following two methods: **signature-based** and **behavior-based**. This work follows the line of behavior-based method, which either statically or dynamically observes the apps behaviors and then inductively categorizes them. More specifically, static analysis techniques observe the unpacked apps statically to identify suspicious trace of data. While useful to detect known threats, it is difficult to identify new malicious apps, such as zero-day attacks [16]. Dynamic analysis techniques, on the other hand, observe the app's behaviors through its actual execution on real devices. Although the information observed correctly reflex the app's exact intention, the execution leads to excessive consumption of Android operating system (OS) [7].

To address these problems, various machine learning methods have recently been studied to sift through large sets of applications and detect malicious applications based on measures of similarity of features [34]. However, several reasons hinder the well adoption of machine learning methods into MD in practice. First, the difficulty of identifying contributed features. To our knowledge, Permissions and Sensitive API calls are still by far the most well-received and -used features for machine learning in MD [24]. Second, few large-scale data sets available to train the machine learning model. Yuan *et al.* [38] utilize only 500 apps in total for training and testing. Third, the complexity of measuring MD scheme has always been a challenge, especially in the case of malware detectors whose authors claim that they work "in the wild". Rasthofer *et al.* [27] claim that their approach has an average precision and recall of more than 92% according to the 10-fold cross validation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASIA CCS'16, May 30 – June 3, 2016, Xi'an, China.

© 2016 ACM. ISBN 978-1-4503-4233-9/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2897845.2897860

[1]http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

[2]http://www.cnbc.com/2015/01/14/mobile-malware-jumped-75-percent-in-2014-report.html

While Allix *et al.* [2] revisit the adequacy of the 10-fold scheme and questions that validating techniques may not perform well in reality. Fourth, the practicality of processing a large-scale data set in a reasonable amount of time. To our knowledge, MassVet [9] is the only tool to compute a large data set within a very short time over Storm.

The aforementioned challenges underline the importance of effective automated malware detection. To this end, this paper presents a combined set of contributed features for machine learning classifiers. We observe roughly 3,000 apps from different sources, analyze their similarities and differences, and identify two unique types of contributed features, sequences, and dynamic behaviors. Together with the two well-known features, requested permissions and sensitive APIs, our approach combines 4 types of contributed features that are uniquely set malicious apps apart.

Furthermore, this paper introduces a streaminglized process that effectively handles the large-scale data set. Our proposed process occurs in three phases: *Preamble*, which prepares resource files for extracting features; *Feature Extraction*, which extracts features from each app based on our combined set of contributed features; and *Classification*, which trains fairly large sets of labeled Android applications and further classifies the data set into different categories, benign and malicious. We extend the idea of "stream" into the execution phases of our process, where all phases happen almost simultaneously with streams of data. With the help of streaming execution phases over data streams, processing a large data set becomes practical.

We have built our streaminglized MD framework, *StormDroid*, on top of Storm, an open-source distributed real-time stream-processing engine that also powers leading web services, such as WebMD, Alibaba, and Yelp. *StormDroid* supports a large-scale analysis of a data stream by a set of worker units that connect to each other, forming a topology. We perform malware detection on roughly 5,000 real apps, a different data set from the 3,000 apps used to observe and analyze contributed features. By comparing the performance of the proposed combination of contributed features against the two traditional features, our approach can improve detection rate by almost 10% in terms of accuracy. Evaluation results show that *StormDroid* can reduce execution time by approximately three times than a single thread, and is able to achieve 94% accuracy, significantly outperforming almost all the top-of-the-line and off-the-shelf antivirus systems.

In summary, this paper presents the following original contributions:

- A novel combination of contributed features observed over a fairly large collection of data that supports our machine learning based MD approach;
- A streaminglized MD framework that supports a large-scale analysis of a data stream in all stages of processing;
- An implementation of feature extraction and data processing, named *StormDroid*, over a distributed real-time stream-processing system;
- A real-word experiment over a fairly large data set (*i.e.*, roughly 8,000 apps), demonstrating the efficiency and scalability of *StormDroid*.

The remainder of the paper is structured as follows. In Section 2, we present the background of Android. We then proceed to overview the learning framework and show our data sets and feature extraction in Section 3. Experimental evaluation is summarized in Section 4. Section 5 discusses the experimental limitation. Section 6 surveys related work. Finally, Section 7 concludes the paper.

2. BACKGROUND

In this section, we provide an overview of the Android application structure, focusing on the important APK resource files. We finally review the two most popular security approaches of Android OS and two common types of malware detection techniques.

2.1 Android Application

Android applications are developed in Java programming language. However, unlike the desktop version of Java apps, the *class* files in Android apps are not loaded directly. Instead, Android's development tools combine the resource file and the *classes.dex* file transformed from *class* file to make an APK; then the APK gets loaded.

APK. The *APK* files are the final outcomes of Android project, thereby they are ready to be registered in the Android markets. The compressed format of this file is *zip*, which allows us to easily open with compressing tools, such as *7-zip, rar*. Note that we cannot figure out the resource file before decompiling.

AndroidManifest. The *Manifest* file is a basic configuration file that holds information about the overall structure of apps. Each app must have an *AndroidManifest.xml* file in its root directory. This file presents essential information about the app to the Android OS and the operating system must have it before running any of the application's code, including version, required permission, etc.

2.2 Security Approaches

Two most popular security approaches of Android OS are listed as follows.

Market Protection. Admittedly, none of the markets can guarantee that the submitted apps are benign. Currently two basic security measures are taken into action to keep a normative environment at the market level, as shown in the following. However, both below measures have proven so far insufficient to combat malware.

- **Signing.** Most markets force developers to legally sign their apps. App signing is the process of digitally signing executables to confirm the app author and guarantee that the code has not been altered or corrupted by using a cryptographic hash. All apps require developers to use a certificate to digitally sign. Once the digital certificate is valid, Android will check the certificate is valid only when the app is installed. Android OS will not be installed without the signature of the app. The signature for each file in the package is processed in order to ensure that the contents of the package is not replaced.
- **Review.** Submitted apps are preliminarily reviewed or analyzed before being downloaded from markets. Once Android markets find a potential malicious app, they will record its signature of the corresponding app for a more in-depth detection later.

Platform Protection. Android platform takes advantage of the following two measures to limit the potential damage of malicious apps once installed on the mobile phone.

- **Sandboxing.** Sandboxing, also called app containerization, is used to limit the environments in which certain code can execute. The goal of sandboxing is to improve security by isolating an app to prevent outside malware, intruders or other apps from interacting with the protected app. The sandboxing has to contain all the files the app needs to execute, which can also create problems between apps that need to interact with one another.
- **Permission.** Android security model highly relies on permission-based method. Before each app being installed, the system will prompt a list of permissions requested by the app and ask the user to confirm the settings for installation. This permission security approach is indeed ineffective as it presents the

risk information of each app in a "stand-alone" fashion and in a way that requires too much technical knowledge and time to distill useful information. For instance, either a benign or a malicious app may require the same permissions and thus it is hard for users to make a right judgment.

2.3 Detection Techniques

Two common types of malware detection techniques, which will frequently be used later, are introduced as follows.

Static analysis. Static analysis attempts to identify malicious code by unpacking and decompiling the app. It is a relatively fast approach and it has been widely used in preliminary analysis to search for blocks of code as signature. Static analysis techniques are well-known in traditional malware detection and have recently gained popularity as efficient mechanisms for market protection, but the result with limited signature database.

Dynamic analysis. Dynamic analysis seeks to identify malicious behaviors after deploying and executing the app on real device. These techniques require some human or automated interaction with the app, as malicious behavior is sometimes triggered only after certain events occur. Dynamic analysis can be used both in the cloud for market protection or directly on the device, though given the uncertainty on the resource consumption and execution time.

3. STORMDROID FRAMEWORK

In this section, we take a first look at the behavior-based learning framework, *StormDroid*, for Android malware detection. We show what it is and how it works. We then follow the workflow that Figure 1 presents and describe their major components and techniques. We finally take an in-depth dive into our data sets and show the methods and steps when selecting the contributed features.

3.1 StormDroid Overview

The high level execution process of our analysis framework, *StormDroid*, as illustrated in Figure 1, occurs in three phases: *Preamble*, which prepares resource files for extracting features; *Feature Extraction*, which extracts features from each app base on our combined set of contributed features; and *Classification*, which trains large sets of labeled Android applications and further classifies the data set into different categories, benign and malicious. We extend the idea of "stream" into the execution phases of our process, where all phases happen almost simultaneously with streams of data. With the help of *streaming execution phases* over *data streams*, processing a large data set becomes practical.

To provide an overview on learning framework, we de facto perform streaminglized fairly large-scale experiments where the training is performed on 3,000 apps. The trained classifier is then used to predict the class of every single application from the training set. Those predictions are then tested on the test data set to estimate the overall accuracy of the selected machine learning approaches. Those predictions are later compared to our reference classification, as shown in Section 4.

3.2 Data Sets

As mentioned earlier, most studies lack a fairly large number of data samples. We fulfill the need by presenting the first fairly large collection of 7,970 Android app samples, including 3,620 malicious samples, which covers the majority of existing to recent ones. Specifically, these 7,970 APK files we collected consist of 4,350 benign apps that are downloaded from Google Play Store, and the other 3,620 malicious APK files where 1,260 have been validated in [40] and the remaining are downloaded from Contagio Mobile Website[3] (360 APKs) and MobiSec Lab Website[4] (2,000 APKs) (See Table 1). Our malicious apps include all varieties of the threats for Android, including phishing, trojans, spyware, root exploits, etc.

3.3 Features

In order to select the contributed features to support the classification results with high precision and sufficient recall, we take an in-depth dive into our features for machine learning. We list four types of features – well received features (required permissions, sensitive API calls), as well as newly-defined features (sequences, dynamic behaviors) – to attempt to characterize each of the android apps by employing both static and dynamic analysis, respectively. All of our features are shown in Table 2. (All features are specified in Appendix A.)

3.3.1 Well-received Features

Permission. Android required permission, extracted from resource code, is commonly used to help detect malicious apps. Each APK has an *AndroidManifest* file in its root directory, which is an essential and important profile including information about the Android application. Android OS must process this profile before it runs any of installation. The profile file declares which permissions the app must have in order to access protected parts of the API and interact with other apps. It also declares the permissions that others are required to have in order to interact with the application's components. These permissions are raw data. We then use *Apktool*[5], a tool for decompiling on APK files, to generate the *AndroidManifest* file and extract requested permissions from it for each APK. To demonstrate that permission settings are indeed relevant to the benign or malware behaviors, we compare top permissions requested by these malicious apps in the data set of 3,032 samples with top permissions requested by benign ones.

The distribution of top 10 permissions requested by 3,032 malicious and benign samples is shown in Figure 2(a). The permissions are ordered by decreasing number of malicious apps. Figure 2(a) clearly shows that "INTERNET", "READ_PHONE_STATE", "ACCESS_NETWORK_STATE", and "WRITE_EXTERNAL_STORAGE" are frequently requested by both benign and malicious apps. In comparison, malicious apps clearly tend to request more frequently in the categories of "READ_SMS", "WRITE_SMS", "SEND_SMS", and "RECEIVE_SMS". Also, in specific top 10 permissions being studied, we observe that malicious apps largely request more permissions than benign ones.

For experiment, we randomly choose 1,516 benign and malicious APKs, respectively from our data sets. We find that several well-known features do not help distinguish between benign and malicious apps, which will increase system overhead. Therefore, we remove some well-known permission-based features, such as "CHANGE_COMPONENT_ENABLED_STATE", and "INTERNAL_SYSTEM_WINDOW". Apart from prior researches, we also add several new features, such as "READ_CALL_LOGS" and "READ_EXTERNAL_STORAGE", as input features. In order to find the top permission features that yield the best performance in detecting new malware, we ultimately select 59 out of 120 permissions as features from the *AndroidManifest* file of each of the 7,970 benign and malicious APKs.

Sensitive API Call. Sensitive API monitoring, based on the reverse engineering, can monitor those sensitive APIs, such as send-

[3]http://contagiominidump.blogspot.hk/

[4]http://www.mobiseclab.org/

[5]http://ibotpeaches.github.io/Apktool/

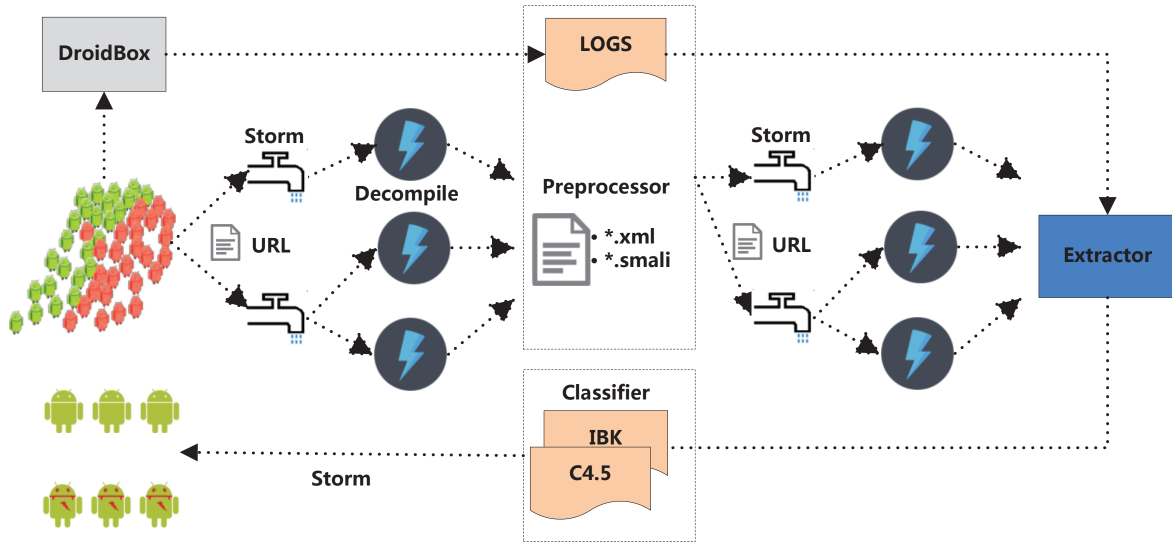


Figure 1: The *StormDroid* Framework for Android Malware Detection

Table 1: Data Sets for Android Malware Detection

Source \ Type of Sets		Universal Set	Analysis Set	Training Set	Test Set	Comparison Set
Benign (APKs)		4,350	1,516	1,500	1,000	0
Malicious (APKs)	**MobiSec Lab**	2,000	900	900	600	500
	Zhou *et al.* [40]	1,260	500	500	300	400
	Contagio	360	116	100	100	100
Total (APKs)		7,970	3,032	3,000	2,000	1,000

Table 2: Features for Machine Learning

Type of Features	Original Features	Selected Features
Permission	120	59
Sensitive API Call	240	90
Sequence	67	1
Dynamic Behavior	15	5
Total	**442**	**155**

ing SMS, accessing user location, device ID, and phone number. We obtain *Smali* files from the static decompiling because converting a *DEX* file to *Smali* files gives us readable code in *Smali* language. We then locate the concrete position of the sensitive API, and embed monitoring *Smali* bytecode to each different sensitive API. We extract sensitive API calls from *Smali* files for each app. The android platform provides a framework API that apps can be used to interact with the underlying Android OS. The framework API consists of a core set of packages and classes. Since most apps use a fairly large number of APIs, we are motivated to use API calls of each app as a feature to characterize and differentiate malware from benign apps.

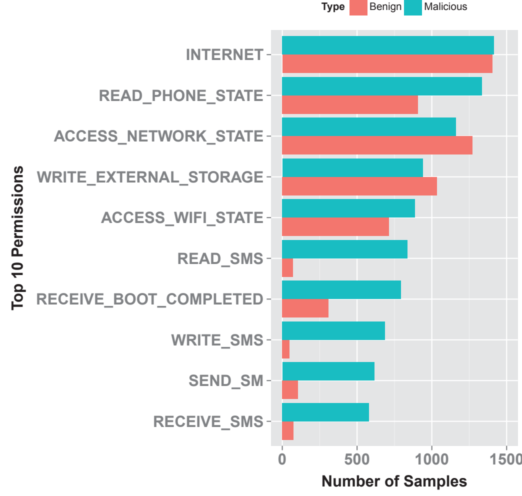
To demonstrate that API calls are indeed helpful for differentiating benign and malware apps, we plot the top 10 sensitive API calls used for both malicious and benign apps in Figure 2(b). As we observe from Figure 2(b), it does not clearly show that malicious apps largely request more API calls than benign ones.

For experiment, we choose 1,516 benign and malicious APKs, respectively from our data sets. We find that several well-known features do not help distinguish between benign and malicious apps, which will increase system overhead. Therefore, we remove some well-known sensitive API call-based features, such as "URLConnection;->getContentType" and "URLConnection;->getURL". In order to find the top sensitive API call features that yield the best performance in detecting new malware, we ultimately select 90 out of 240 sensitive API calls as features from the *Smali* file of each of the 7,970 benign and malicious APKs.

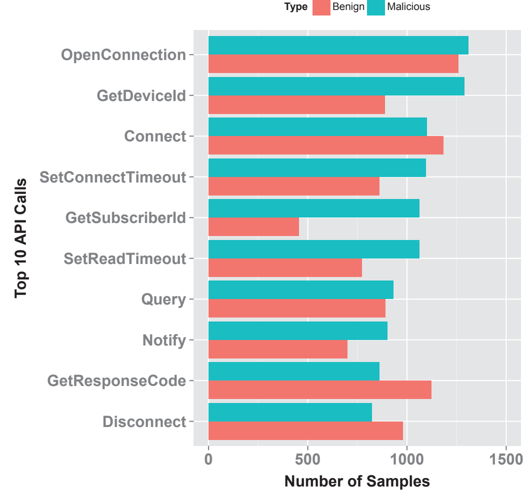
3.3.2 Newly-defined Features

Sequence. Through closely examining the sensitive API calls collected over our fairly large collection of apps, we notice that the malicious apps tend to have drastically different sensitive API calls, which supports the assumption that malicious apps are distinguishable from benign apps. Therefore, we take a closer look at the quantity of sensitive API calls, and novelly define "Sequence" as features extracted from *Smali* files by recording the number of sensitive API calls requested by the malicious apps and the benign ones, respectively. We propose three quantitive metrics for extracting sequence, as depicted in the following.

(i) "Subtraction-Differential" metric. In order to analyze the difference between malicious apps and benign ones, we first define a "subtraction-differential" metric. The differential $d_{(s,m,b)}$ (or $d_{(s,b,m)}$) depends on a particular sensitive API call s, a mali-



(a) Comparison of top 10 requested permissions by 3,032 benign and malicious apps



(b) Comparison of top 10 requested sensitive API calls by 3,032 benign and malicious apps

Figure 2: Comparison of top 10 requested permissions and sensitive API calls by 3,032 benign and malicious apps

cious app m, a benign app b, and is defined as follows:

$$m_s = \text{a malicious app } m \text{ w.r.t. \# sensitive API call } s;$$
$$b_s = \text{a benign app } b \text{ w.r.t. \# sensitive API call } s;$$
$$d_{(s,m,b)} = \text{difference between } m_s \text{ and } b_s;$$
$$d_{(s,b,m)} = \text{difference between } b_s \text{ and } m_s.$$

We then denote $\mathcal{D}_1$ (resp. $\mathcal{D}_2$) as the set of top values of $d_{(s,m,b)}$ (resp. $d_{(s,b,m)}$) that outnumber the threshold 200, where the cardinality of the set $\mathcal{D}_1$ (resp. $\mathcal{D}_2$) is 13 (resp. 14). We finally denote the union $\mathcal{D} = \mathcal{D}_1 \cup \mathcal{D}_2$.

(ii) "Logarithm-Differential" metric. Followed up with the above definitions, we further define a "logarithm-differential" metric, as shown as follows:

$$\lg\left\{\left(\frac{m_s}{b_s+1}\right)+1\right\}. \quad (1)$$

Notice that Equation (1) has already been regularized to avoid ambiguity.

We filter out the top 16 values of Equation (1) that are greater than 0.4 (symbolized as the set $\mathcal{L}_1$) and the bottom 11 values of Equation (1) that are less than 0.05 (symbolized as the set $\mathcal{L}_2$). We finally denote the union $\mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2$.

(iii) "Subtraction-Logarithm" metric. We eventually denote set $\mathcal{S}$ to be the intersection $\mathcal{S} = \mathcal{D} \cap \mathcal{L}$, where the cardinality of the set $\mathcal{S}$ is 13. We summarize the details of all the sets in Table 3.

For experiment, we set weights for each element in the sequence set $\mathcal{S}$ depending on the "Subtraction-Differential" metric. For example, if the $d_{(s,m,b)}$ of the first feature "*TelephonyManager;->getSimState*" in set $\mathcal{S}$ is 607, we then set its weight as $+\left({}^{607}/_{1{,}516}\right)$; if the $d_{(s,b,m)}$ of the last feature "*DownloadManager;->enqueue*" in set $\mathcal{S}$ is 200, we then set its weight as $-\left({}^{200}/_{1{,}516}\right)$. Furthermore, we also set a counting flag – *sum* – in set $\mathcal{S}$. To be specific, if the APK contains at least one of the features either in set $\mathcal{D}_1 \cap \mathcal{L}_1$ or in set $\mathcal{D}_2 \cap \mathcal{L}_2$, we add $+\left({}^{d_{(s,m,b)}}/_{1{,}516}\right)$ or $-\left({}^{d_{(s,b,m)}}/_{1{,}516}\right)$ to *sum*, respectively; if the APK does not contain any one of the features in set $\mathcal{S}$, the value of *sum* remains unchanged.

Table 3: The Sequences of Apps

Set	# Sequences by "Subtraction-Differential"	# Sequences by "Logarithm-Differential"
$\mathcal{D}_1$	13	N/A
$\mathcal{D}_2$	14	N/A
$\mathcal{D}$	27	N/A
$\mathcal{L}_1$	N/A	16
$\mathcal{L}_2$	N/A	11
$\mathcal{L}$	N/A	27
$\mathcal{S}$	13	

By further tuning the parameters, we conclude that for each of the 7,970 apps if the *sum* value of the set $\mathcal{S}$ outnumber the threshold 0.4, the corresponding sequence is heuristically marked as '1'; otherwise, it is marked as '0'. Hence, we convert such a 13-dimensional input vector into one sequence feature of the set $\mathcal{S}$.

Figure 3 (resp. Figure 5) shows the top 13 (Malicious - Benign) (resp. top 14 Benign - Malicious) sequences sorted by 3,032 malicious and benign samples. Correspondingly, the quantity of their exact differences are represented in Figure 4 (resp. Figure 6). The sequences are ordered by decreasing number of differences, respectively.

Dynamic Behavior. Dynamic behaviors observe the malicious activities triggered by each application. Based on our observation that malicious apps tend to request sensitive information more frequently than the benign one, we thus include it into our combinatorial set of contributed features. To obtain the dynamic behaviors of Android app, we run the *apk* file of an app in *DroidBox*[6]. We then statically analyze the saved log files to extract the top features of dynamic behaviors. That would be network activity, file system access, interaction with the operating system, etc. As shown

[6]https://code.google.com/p/droidbox/

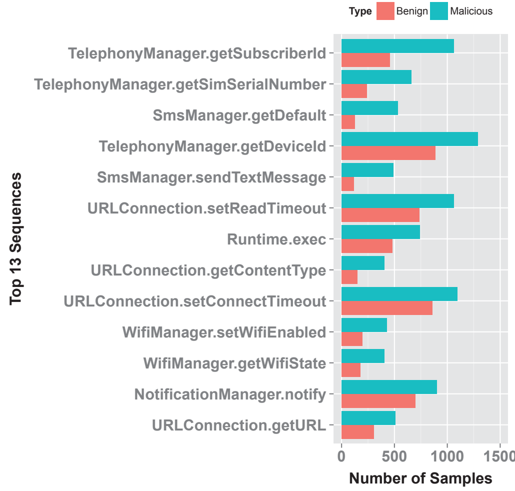


Figure 3: Top 13 differences of sensitive API calls between malicious and benign apps by 3,032 benign and malicious samples

Figure 5: Top 14 differences of sensitive API calls between benign and malicious apps by 3,032 benign and malicious samples

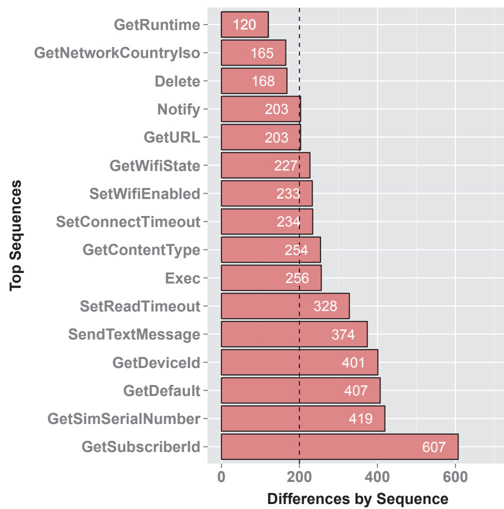


Figure 4: The differences of sensitive API calls between malicious and benign apps sorted by 3,032 benign and malicious apps

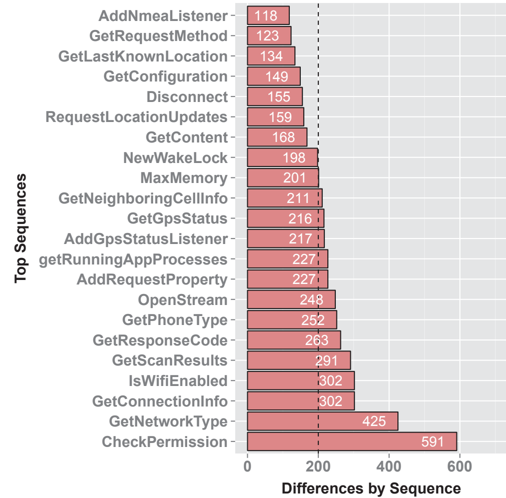


Figure 6: The differences of sensitive API calls between malicious and benign apps sorted by 3,032 benign and malicious apps

in Figure 7, *accessfiles* is frequently requested by both benign and malicious apps.

For experiment, we choose 1,516 benign and malicious APKs, respectively from our data sets. Likewise, we remove some well-known dynamic behavior-based features, such as "dexclass_load" and "fdaccess". In order to find the top dynamic behavior features that yield the best performance in detecting new malware, we ultimately select 5 out of 15 dynamic behaviors as features from the logs of each of the 7,970 benign and malicious APKs.

4. EXPERIMENTAL EVALUATION

The goals of our experiments are to determine: (i) the capabilities of defined contributed features; (ii) the capabilities of accurately detecting malicious apps; and (iii) the efficiency and scalability of real-time analysis.

4.1 Experimental Setup

We take advantage of the different types of features to obtain two groups of experimental result. A group uses a basic set of contributed features as a benchmark, which includes the two widely used features, namely requested permissions and sensitive API calls; and the other group is joined by our defined two types of contributed features, which are sequences and dynamic behaviors, to determine if they are indeed contributing to differentiate the malicious apps. To do so, we compare two sets of contributed features over the same data set, for different machine learning classifiers.

For a meaningful comparison, we list the results of two feature

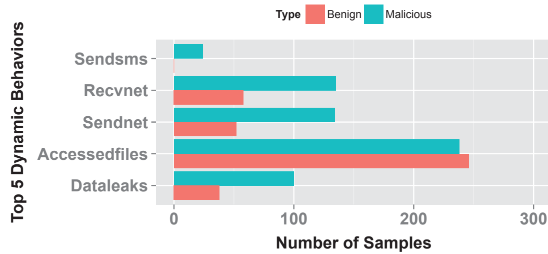


Figure 7: Comparison of top 5 requested dynamic behaviors by 3,032 benign and malicious apps sorted by 3,032 benign and malicious apps

groups which are used to train on six classifiers with respect to the aspects of true positives (denoted as TP), false positives (denoted as FP), receiver operating characteristic (denoted as ROC), precision (denoted as P), recall rate (denoted as Rec), and accuracy (denoted as A). TP rate refers to the malicious instances that are classified as malicious apps relative to all malicious instances. FP rate refers to all non-malicious instances that are classified as malicious apps relative to all malicious instances. ROC is a graphical depiction of classifier performance that shows the trade-off between increasing TP rates and increasing FP rates as the discrimination threshold of the classifier is varied. Precision rate refers to the probability that an app is correctly classified as a malicious app. Recall rate is defined as the portion of the total malicious apps that are classified as malware. Accuracy simply measures that the classifier makes the correct prediction. We highlight the best result of the six classifiers in bold.

We randomly select 1,000 malware as samples out of the set of 3,620 malicious samples and scan them using *StormDroid* and other industrial malware detecting tools. For comparison, the 1,000 samples include three sections as shown in Table 1. In order to maintain objectivity, the 1,000 samples contain both benchmarks before 2013 (partially from [40]) and most recent sets, more than half of which are the latest malware.

We measure the efficiency and scalability of *StormDroid* performance, and perform the entire process of *StormDroid*, using single thread and distributed real-time streaming, respectively, on a server with 16 GB memory, 8 cores at 3.0 GHz, and 1 TB hard drives. We randomly choose 200 out of 7,970 APKs with a single thread to compare with *StormDroid*. The 200 APKs are in four groups with the number of 200, 50, 40, 20 in each group, respectively.

4.2 Classification Methods

In this section, we analyze the performance of different machine learning classifiers, including Support Vector Machines (SVM), Decision Tree (C4.5), Artificial Neural Networks (MLP), Naive Bayes (NB), K-Nearest Neighbors (IBK), and Bagging predictor. Our goal is to determine whether the selected features – permissions, sensitive API calls, sequences, and dynamic behaviors – can provide insights into characterizing the behaviors of apps.

In machine learning, SVM is supervised learning models with associated learning algorithms that analyze data and recognize patterns, used for classification and regression analysis. C4.5, as an extension of earlier ID3 algorithm, can be used for classification. MLP are a family of statistical learning models inspired by biological neural networks and are used to estimate or approximate functions that can depend on a fairly large number of inputs. NB classifiers are a family of simple probabilistic classifiers based on applying Bayes' theorem with strong independence assumptions between the features. IBK is a non-parametric method usually used for classification and regression.

The training set of each base classifier is generated by randomly drawing training samples, with replacement, from the original training set (See Table 5). The final prediction of an instance is based on the majority vote of all base classifiers' predictions.

4.3 Experimental Results

(i) The capabilities of defined contributed features.

We use *StormDorid*, which has already packaged all these aforementioned machine learning algorithms, to test the level of accuracy with respect to different algorithms. We list their average values of each index. Table 4 reports the classification results with respect to different classifiers on all benchmark data sets. As shown in Table 4, the second category generally outperforms the first category. Namely, the combined features yield the best outcomes. Thus, we choose the combination of apps permissions, sensitive API calls, sequences, and dynamic behaviors as our feature selection scheme.

Moreover, Table 4 reports the classification results across different categories. In the first category, the performance of K-Nearest Neighbors (IBK) and Artificial Neural Networks (MLP) classifiers are better than other classifiers; and their accuracy rates are larger than 91.00%. The accuracy of the Bagging predictor (90.65%) and Support Vector Machines (SVM) classifier (90.46%) are fairly good. In the second category, K-Nearest Neighbors (IBK) and Support Vector Machines (SVM) classifier achieve their best accuracy rates at 93.80% and 93.20%, respectively, which is 1.22% and 2.74% higher than its counterparts in the first category. Hence, the K-Nearest Neighbors (IBK) classifier deliver the best accuracy by using our selected features.

Furthermore, we evaluate the execution time for each classifier. Support Vector Machines (SVM), Decision Tree (C4.5), K-Nearest Neighbors (IBK), Naive Bayes (NB), and Bagging predictor complete the experiments by using almost the same amount of time, *i.e.*, less than 60 seconds. However, the Artificial Neural Networks (MLP) takes significantly more time than the above-mentioned ones as multi-layer neural networks are time-consuming.

We further compare our work with the previous work. As shown in Table 5, our accuracy rate (all greater than 90.00%) completely outperforms the accuracy rate (no greater than 80.00%) in Yuan *et al.* [38]. We achieve the highest accuracy because of the novel feature combination method. That is, we select the combination of four top features – apps permissions, sensitive API calls, sequences, and dynamic behaviors – as our features. In contrast, the previous work only use combined features, yet without sequence, as feature groups.

Furthermore, the best performing classifier in our work is the K-Nearest Neighbors (IBK), which requires little computational resources. By choosing a good set of features and the appropriate classifier, we can complete classification process less than one minute.

(ii) The capabilities of accurately detecting malicious apps.

To circumvent the over-fitting issue and to better understand the coverage of our approach, we randomly sample 1,000 malware apps from our data set and scan them using *StormDroid* and other well-known industrial malware detection tools, such as Trend Micro and Kaspersky. The coverage of *StormDroid*, with the combined top features, is 94.60%, better than what can be achieved by any individual scanner, including such top-of-the-line antivirus systems as 360 (86.20%), McAfee (84.20%), Avira (75.40%), Kasper-

Table 4: The performance of machine learning algorithms

ML Algorithm	Category 1 (Permissions + Sensitive API calls)	Category 2 (Permissions + Sensitive API calls + Sequences + Dynamic Behaviors)
Support Vector Machines (SVM)	90.50%(TP) 9.50%(FP) 90.40%(ROC) 90.50%(P) 90.50%(Rec) 90.46%(A)	93.20%(TP) 6.80%(FP) 93.20%(ROC) 93.20%(P) 93.20%(Rec) 93.20%(A)
Decision Tree (C4.5)	88.40%(TP) 11.60%(FP) 92.00%(ROC) 88.20%(P) 92.00%(Rec) 88.24%(A)	91.00%(TP) 9.00%(FP) 92.10%(ROC) 91.40%(P) 91.00%(Rec) 91.00%(A)
Artificial Neural Networks (MLP)	91.30%(TP) 8.70%(FP) 96.20%(ROC) 91.40%(P) 91.20%(Rec) 91.23%(A)	92.60%(TP) 7.40%(FP) 97.30%(ROC) 92.60%(P) 92.60%(Rec) 92.60%(A)
Naive Bayes (NB)	88.20%(TP) 11.80%(FP) 94.20%(ROC) 88.90%(P) 88.20%(Rec) 88.15%(A)	90.80%(TP) 9.20%(FP) 94.20%(ROC) 91.10%(P) 90.80%(Rec) 90.80%(A)
K-Nearest Neighbors (IBK)	92.60%(TP) 7.40%(FP) 94.90%(ROC) 92.60%(P) 92.60%(Rec) 92.58%(A)	**93.80%(TP) 6.20%(FP) 96.70%(ROC) 93.80%(P) 93.80%(Rec) 93.80%(A)**
Bagging predictor	90.70%(TP) 9.30%(FP) 96.80%(ROC) 90.70%(P) 90.70%(Rec) 90.65%(A)	92.80%(TP) 7.20%(FP) 97.90%(ROC) 92.80%(P) 92.80%(Rec) 92.80%(A)

Table 5: Comparative results of our work and the previous work

ML Algorithm	Yuan *et al.* [38] (Accuracy)	Ours (StormDroid) (Accuracy)
Support Vector Machines (SVM)	**80.00%**	93.20%
Decision Tree (C4.5)	77.50%	91.00%
Artificial Neural Networks (MLP)	79.50%	92.60%
Naive Bayes (NB)	79.00%	90.80%
K-Nearest Neighbors (IBK)	N/A	**93.80%**
Bagging predictor	N/A	92.80%
Best Performing Classifier	**Support Vector Machines (SVM)**	**K-Nearest Neighbors (IBK)**
Universal Data Set Size	500 APKs	7,970 APKs
Training Set Size	300 APKs	3,000 APKs
Test Set Size	200 APKs	2,000 APKs

sky (55.60%), and Trend Micro (41.40%). The details of the reference experiment study is presented in Table 6.

(iii) The efficiency and scalability of real-time analysis.

To support a high-performance detection of apps, *StormDroid* is designed to run on top of a stream processing framework (See Figure 1). Specifically, our implementation is built on top of the *Storm*[7], an open-source distributed real-time stream-processing engine that also powers leading web services, such as WebMD, Alibaba, and Yelp. *StormDroid* supports a large-scale analysis of a data stream by a set of worker units that connect to each other, forming a topology. In our implementation, the work flow of the whole detection process is converted into such a topology: a submitted app is first disassembled to extract its features; then, the app's "differential" metrics are calculated, the intersection analysis is run, and a binary input vector is finally obtained. Each operation here is delegated to a worker unit on the topology and all the data associated with the app are in a single stream. The *StormDroid* engine is designed to support concurrently processing multiple streams, which enables a market to efficiently detect a large number of submissions.

Running on top of the *Storm* stream processor, our prototype is tested against the data sets divided into different test groups with the same size, in order to show that the size of the group is unaffected with the efficiency of *StormDroid*. We ultimately choose 200 out of 7,970 APKs with a single thread to test the efficiency of *StormDroid*. Table 7 clearly shows that the execution efficiency of *StormDroid* significantly outperforms the single thread by approximately three times according to each respective group.

[7]http://storm.apache.org/

As shown in Table 7, the smaller the data set is, the larger the ratio of the running time of *StormDroid* relative to that of the single thread becomes. Overall, we show that the size of the group is unaffected with the efficiency enhancement and the *StormDroid* is indeed capable of scaling to the massive data sets.

5. DISCUSSION

(i) Blurred line between benign and malicious. In practice, the line between benign and malicious might be blurred and subjective. It depends on specific security requirements and use cases to determine whether an access pattern is really benign or malicious. For example, individual users may like rooting their own devices and using the game hacking apps mentioned above, while game developers treat them as malicious because they bypass the in-app purchase.

(ii) The unavailability of representative malicious and benign applications. To avoid crafting detection patterns manually, we make use machine learning for generating detection models. While learning techniques provide a powerful tool for automatically inferring models, they require a representative basis of data for training. That is, the quality of the detection model of *StormDroid* critically depends on the availability of representative malicious and benign applications. While it is straightforward to collect benign applications, gathering recent malware samples requires some technical effort.

(iii) The limitation of decompilation. Decompiling is the preparation of extracting features of Android malware, the mainstay of training potentially harmful mobile apps. However, we cannot guarantee that all the APKs can be decompiled successfully. We

Table 6: Reference experiment: The coverage of other leading malware detection tools

Malware Detection Tool	The Number of Detection	The Coverage of Detection (Percentage)
Ours (StormDroid)	1,000	**94.60%**
Trend Micro	1,000	41.40%
Kaspersky	1,000	55.60%
360	1,000	86.20%
McAfee	1,000	84.20%
Avira	1,000	75.40%

Table 7: Experimental evaluation

# APKs	Experimental Times	# APKs per Group	AVG Single Time (sec)	AVG StromDroid Time (sec)	Ratio
200	10	20	715	209	0.29
200	10	40	718	201	0.28
200	10	50	712	203	0.29
200	10	200	710	207	0.29

***Ratio is defined as the AVG *StromDroid* Time relative to the AVG Single Time, *i.e.*,** $^{AVGStromDroidTime}/_{AVGSingleTime}$.

find that the decompilation to source code is more likely to fail than to *Smali* files, which hinders the feature extraction process. Therefore, we alternatively choose to decompile the APKs to *Smali* files. **(iv) The inexplicability of our features and the misuse of machine learning.** In this paper, we use four types of features – required permissions, sensitive API calls, sequences, and dynamic behaviors – for machine learning. In the course of feature selection, what we de facto do is to simply compare different features of training apps (those including malicious and benign apps) to inspect their different parts to identify suspicious features. These features, once found to be inexplicable, are almost certain to be malicious, as discovered in our study (See Section 3). This similarity analysis is well suited for finding previously unknown malicious behaviors, without resorting to any complex techniques. For example, apart from prior researches, we surprisingly observe that "URLConnection;->getContentType" and "URLConnection;->getURL" are newly-discovered top suspicious features for malicious behaviors. In addition, for example, the screen-locking extortion behavior can be detected usually by simultaneously discovering a series of permissions, which are "RECEIVE_SYSTEM_ALERT_WINDOW", "SYSTEM_ALERT_WINDOW", and "WEKA_LOCK". In this work, by training the combination of our newly-defined and well-received features, we are able to discover the Android lock-screen-type ransomware. As a proof-of-concept, we have novelly attempted to define many different features and extracted 155 features for our detection model. In the future, we, nevertheless, will also argue that the malware detection is still able to be effective even when using a small number of features and simple detection algorithms ensure that stringent resource constraints (*i.e.*, CPU and battery) on the device are met.

Another limitation which follows from the use of machine learning is the possibility of mimicry and poisoning attacks [26]. Such attempts to escape detection are likely to be deemed suspicious and may invite further scrutiny. While obfuscation strategies, such as repackaging, code reordering, or junk code insertion do not affect *StormDroid*, renaming of activities and components between the learning and detection phase may impair discriminative features [29]. Similarly, an attacker may succeed in lowering the detection score of *StormDroid* by incorporating benign features or fake invariants into malicious applications [26]. Although such attacks against learning techniques cannot be ruled out in general, the thorough sanitization of learning data and a frequent retraining on representative data sets can limit their impact.

6. RELATED WORK

Thwarting malware attacks on smart devices has recently become a long-standing topic.

Two generic approaches that have been proposed to detect malware: **static analysis** [12, 16, 17, 21, 25] and **dynamic analysis** [6, 10, 13, 33]. Although several approaches have been successfully put into practice, Moser *et al.* [22] developed various obfuscation techniques that are especially effective against static analysis. On the other hand, approaches based on dynamic code analysis [11] are promising, which provides a complete overview of automated dynamic malware analysis techniques, but adopting and adapting them to smart devices is not straightforward. Specifically, power consumption and a constant monitoring executed on the platform may simply be unaffordable [20]. External analysis performed on the cloud in near real time constitute an alternative, though it is not exempted from privacy-pertinent leakages.

Therefore, resource limitations of smartphones have lead researchers to propose collaborative analysis techniques, where the analysis is made by a network of devices. Both static and dynamic analysis [30] have been proposed using these techniques. Static and dynamic analysis solutions are primarily implemented using two methods: **signature-based** [3, 14, 18, 39, 41] and **behavior-based** [13, 28, 35, 37]. Signature-based is a common method used by antivirus vendors and it relies on the identification of unique signatures that define the malware. While being very precise, signature-based methods are useless against unknown malicious code. The behavior-based methods are based on rules or features which are either determined by experts or by machine learning techniques that define a malicious or a benign behavior, in order to detect unknown malware [27].

As recently proposed by Antivirus companies, static analysis can be deployed for malware detection in Android devices [32]. But due to the limited resources of smartphones, most of the recent proposals for malware detection on Android devices are based on behavior analysis for anomaly detection.

The difficulty of manually crafting and updating detection patterns for Android malware has motivated the application of machine learning. Several methods have been proposed that analyze applications automatically using learning methods [4, 25, 31]. As

an example, the method of Peng *et al.* [25] applied probabilistic learning methods to the permissions of applications for malware detection. Zhou *et al.* [41] first proposed to use permission behavior to detect new Android malware and then applies heuristic filtering for detecting unknown Android malware. This hybrid method, called DroidRanger, resolves the disadvantage of lacking ability to detect unknown malware. Similarly, the methods Crowdroid [6], DroidMat [36], Adagio [15], MAST [8], and DroidAPIMiner [1] analyzed features statically extracted from Android applications using machine learning techniques. All these existing methods have essentially advanced the Android malware detection, but the misuse detection is not adaptive to the novel Android malware and always requires frequent updating of the signatures. However, our work differs from aforementioned approaches by novelly proposing a framework to analyze Android Apps based on machine learning techniques. The framework relies on a combination of requested permissions, sensitive API calls, malicious execution sequences, and dynamic behaviors, which extracts features and builds classifiers to detect malicious apps before installation.

From another perspective, in order to improve the smartphone security and privacy, a number of platform-level extensions have been proposed. Specifically, Apex [23], MockDroid [5], TISSA [42] and AppFence [19] extended the current Android framework to provide find-grained controls of system resources accessed by untrusted third-party applications. Note that none of them characterizes the existing Android malware. Recently, MassVet [9] was developed for vetting apps at a massive scale within a very short time by also using *Storm*. Their approach simply compares a submitted app with all those already on a market, focusing on the difference between those sharing a similar UI structure and the commonality among those seemingly unrelated. However, to the best of our knowledge, followed up with the line of behavior-based methods, we are the first to develop a streaminglized machine learning-based real-time analysis system, *StormDroid*, making it fast to reliably process large streams of data when extracting features and running machine learning classifiers.

7. CONCLUSION

We proposed a streaminglized machine learning framework for malware detection. First, we collected two unique type of contributed features that are observed over a randomly selected large-scale data set, and introduced a novel combination set of contributed features for machine learning. Second, we streaminglized the whole process of malware detection by streaming not only the data processing but all stages of feature extraction and classification. The framework, *StormDroid*, demonstrated its accuracy and efficiency in classifying malicious applications. Indeed, evaluation results showed that *StormDroid* is able to achieve 94% accuracy, significantly outperforming almost all the top-of-the-line and off-the-shelf antivirus systems. This work not only provides novel contributed features for malware detection, but also suggests a new perspective on the streaminglized MD process. We expect our study to draw more public attention to this serious security issue and hopefully motivate better detection methods, as shown in this paper. Only time will tell whether malicious apps can be detected on a massive scale, thereby rendering the Google Play Store not only productive but also completely privacy-preserving.

Acknowledgements

This work was supported in part by the National Natural Science Foundation of China, under Grant 61502170, 61272444, 61411146001, U1401253, and U1405251, in part by the Science and Technology Commission of Shanghai Municipality under Grant 13ZR1413000, in part by Pwnzen Infotech Inc.

8. REFERENCES

[1] Y. Aafer, W. Du, and H. Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *Security and Privacy in Communication Networks*, pages 86–103. Springer, 2013.

[2] K. Allix, T. F. Bissyandé, Q. Jérome, J. Klein, Y. Le Traon, et al. Large-scale machine learning-based malware detection: confronting the 10-fold cross validation scheme with reality. In *Proceedings of the 4th ACM conference on Data and application security and privacy*, pages 163–166. ACM, 2014.

[3] 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 Notices*, volume 49, pages 259–269. ACM, 2014.

[4] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 73–84. ACM, 2010.

[5] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, pages 49–54. ACM, 2011.

[6] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26. ACM, 2011.

[7] L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 143–163. Springer, 2008.

[8] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. Mast: triage for market-scale mobile malware analysis. In *Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks*, pages 13–24. ACM, 2013.

[9] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *USENIX Security*, volume 15, 2015.

[10] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra. Madam: A multi-level anomaly detector for android malware. In *MMM-ACNS*, volume 12, pages 240–253. Springer, 2012.

[11] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(2):6, 2012.

[12] K. O. Elish, D. D. Yao, B. G. Ryder, and X. Jiang. A static assurance analysis of android applications. 2013.

[13] 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 Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[14] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 576–587. ACM, 2014.

[15] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 45–54. ACM, 2013.

[16] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 281–294. ACM, 2012.

[17] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *NDSS*, 2012.

[18] K. Griffin, S. Schneider, X. Hu, and T.-C. Chiueh. Automatic generation of string signatures for malware detection. In *Recent Advances in Intrusion Detection*, pages 101–120. Springer, 2009.

[19] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 639–652. ACM, 2011.

[20] H. Kim, J. Smith, and K. G. Shin. Detecting energy-greedy anomalies and mobile malware variants. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, pages 239–252. ACM, 2008.

[21] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 229–240. ACM, 2012.

[22] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*, pages 421–430. IEEE, 2007.

[23] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 328–332. ACM, 2010.

[24] N. Peiravian and X. Zhu. Machine learning for android malware detection using permission and api calls. In *Tools with Artificial Intelligence (ICTAI), 2013 IEEE 25th International Conference on*, pages 300–305. IEEE, 2013.

[25] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Using probabilistic generative models for ranking risks of android apps. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 241–252. ACM, 2012.

[26] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif. Misleading worm signature generators using deliberate noise injection. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–pp. IEEE, 2006.

[27] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *2014 Network and Distributed System Security Symposium (NDSS)*, 2014.

[28] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: automatic security analysis of smartphone applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 209–220. ACM, 2013.

[29] V. Rastogi, Y. Chen, and X. Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 329–334. ACM, 2013.

[30] S. Rosen, Z. Qian, and Z. M. Mao. Appprofiler: a flexible method of exposing privacy-related behavior in android applications to end users. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 221–232. ACM, 2013.

[31] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Android permissions: a perspective combining risks and benefits. In *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, pages 13–22. ACM, 2012.

[32] A.-D. Schmidt, R. Bye, H.-G. Schmidt, J. Clausen, O. Kiraz, K. Yüksel, S. Camtepe, S. Albayrak, et al. Static analysis of executables for collaborative malware detection on android. In *Communications, 2009. ICC'09. IEEE International Conference on*, pages 1–5. IEEE, 2009.

[33] 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*, 38(1):161–190, 2012.

[34] G. Tahan, L. Rokach, and Y. Shahar. Mal-id: Automatic malware detection using common segment analysis and meta-features. *The Journal of Machine Learning Research*, 13(1):949–979, 2012.

[35] C. Wu, Y. Zhou, K. Patel, Z. Liang, and X. Jiang. Airbag: Boosting smartphone resistance to malware infection. In *Proceedings of the Network and Distributed System Security Symposium*, 2014.

[36] 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 *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*, pages 62–69. IEEE, 2012.

[37] L.-K. Yan and H. Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *USENIX security symposium*, pages 569–584, 2012.

[38] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue. Droid-sec: deep learning in android malware detection. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 371–372. ACM, 2014.

[39] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, scalable detection of piggybacked mobile applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 185–196. ACM, 2013.

[40] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.

[41] 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 *NDSS*, 2012.

[42] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming information-stealing smartphone applications (on android). In *Trust and Trustworthy Computing*, pages 93–107. Springer, 2011.

APPENDIX

A. LIST OF FEATURES

Table 8: List of Features

Permission	*WRITE_SYNC_SETTINGS	*LocationManager.clearTestProviderLocation
* ACCESS_COARSE_LOCATION	*WRITE_EXTERNAL_STORAGE	*LocationManager.clearTestProviderEnabled
* ACCESS_FINE_LOCATION	**Sensitive API Call**	*LocationManager.addTestProvider
* LOCATION_EXTRA_COMMANDS	*URL.openConnection	*LocationManager.addProximityAlert
* ACCESS_MOCK_LOCATION	*URL.openStream	*ContentResolver.applyBatch
* ACCESS_NETWORK_STATE	*URL.getConten	*ContentResolver.bulkInsert
* ACCESS_WIFI_STATE	*TelephonyManager.getCallState	*ContentResolver.openAssetFileDescriptor
* AUTHENTICATE_ACCOUNTS	*TelephonyManager.getDeviceId	*ContentResolver.openFileDescriptor
* BATTERY_STATS	*TelephonyManager.getDeviceSoftwareVersion	*ContentResolver.query
* BLUETOOTH	*TelephonyManager.getNeighboringCellInfo	*ContentResolver.registerContentObserver
* BLUETOOTH_ADMIN	*TelephonyManager.getNetworkCountryIso	*ContentResolver.update
* BROADCAST_STICKY	*TelephonyManager.getNetworkOperator	*ContentResolver.delete
* CALL_PHONE	*TelephonyManager.getNetworkOperatorName	*Runtime.getRuntime
* CAMERA	*TelephonyManager.getNetworkType	*Runtime.exec
* CHANGE_CONFIGURATION	*TelephonyManager.getPhoneType	*Runtime.addShutdownHook
* CHANGE_NETWORK_STATE	*TelephonyManager.getSimCountryIso	*Runtime.getRuntime
* CHANGE_WIFI_MULTICAST_STATE	*TelephonyManager.getSimSerialNumber	*Runtime.maxMemory
*CHANGE_WIFI_STATE	*TelephonyManager.getSimState	*URLConnection.addRequestProperty
*CLEAR_APP_CACHE	*TelephonyManager.getSimOperator	*URLConnection.connect
*DELETE_PACKAGES	*TelephonyManager.getSimOperatorName	*URLConnection.getContent
*DEVICE_POWER	*TelephonyManager.getSubscriberId	*URLConnection.getContentType
*DISABLE_KEYGUARD	*TelephonyManager.isNetworkRoaming	*URLConnection.getURL
*EXPAND_STATUS_BAR	*SmsManager.divideMessage	*URLConnection.setConnectTimeout
*FLASHLIGHT	*SmsManager.getDefault	*URLConnection.setReadTimeout
*GET_ACCOUNTS	*SmsManager.sendMultipartTextMessage	*ActivityManager.getLargeMemoryClass
*GET_PACKAGE_SIZE	*SmsManager.sendTextMessage	*ActivityManager.getRunningAppProcesses
*GET_TASKS	*HttpURLConnection.disconnect	*ActivityManager.killBackgroundProcesses
*INSTALL_PACKAGES	* HttpURLConnection.getContentEncoding	*ActivityManager.restartPackage
*INTERNET	* HttpURLConnection.getRequestMethod	*System.getConfiguration
*MANAGE_ACCOUNTS	*HttpURLConnection.getResponseCode	*System.getUriFor
*MODIFY_AUDIO_SETTINGS	*HttpURLConnection.getResponseMessage	*BluetoothAdapter.enable
*MODIFY_PHONE_STATE	*PowerManager.newWakeLock	*DownloadManager.enqueue
*PROCESS_OUTGOING_CALLS	*PowerManager.isScreenOn	*DownloadManager.query
*READ_CALENDAR	*PackageManager.checkPermission	*LocationManager.addGpsStatusListener
*READ_CALL_LOG	*NotificationManager.notify	*LocationManager.addNmeaListener
*READ_CONTACTS	*NotificationManager.cancel	**Sequence**
*READ_EXTERNAL_STORAGE	*TelephonyManager.getVoiceMailNumber	*TelephonyManager;->getSubscriberId
*READ_LOGS	*WifiManager.setWifiEnabled	*TelephonyManager;->getSimSerialNumber
*READ_PHONE_STATE	*WifiManager.saveConfiguration	*SmsManager;->getDefault
*READ_SMS	*WifiManager.removeNetwork	*SmsManager;->sendTextMessage
*READ_SYNC_SETTINGS	*WifiManager.isWifiEnabled	*Runtime;->exec
*RECEIVE_BOOT_COMPLETED	*WifiManager.getWifiState	*URLConnection;->getContentType
*RECEIVE_MMS	*WifiManager.getScanResults	*WifiManager;->setWifiEnabled
*RECEIVE_SMS	*WifiManager.getDhcpInfo	*WifiManager;->getWifiState
* RECEIVE_WAP_PUSH	*WifiManager.getConnectionInfo	*URLConnection;->getURL
*RECORD_AUDIO	*WifiManager.getConfiguredNetworks	*LocationManager;->addGpsStatusListener
*REORDER_TASKS	*WifiManager.enableNetwork	*LocationManager;->getGpsStatus
*RESTART_PACKAGES	*WifiManager.createMulticastLock	*TelephonyManager;->getNeighboringCellInfo
*SEND_SMS	*WifiManager.createWifiLock	*Runtime;->maxMemory
*SET_WALLPAPER	*WifiManager.calculateSignalLevel	**Dynamic Behavior**
*SYSTEM_ALERT_WINDOW	*WifiManager.addNetwork	*Sendsms
*USE_CREDENTIALS	*LocationManager.sendExtraCommand	*Recvnet
* VIBRATE	*LocationManager.requestLocationUpdates	*Sendnet
*WAKE_LOCK	* LocationManager.getLastKnownLocation	* Accessedfiles
* WRITE_APN_SETTINGS	*LocationManager.getGpsStatus	*Dataleaks
*WRITE_CALENDAR	*LocationManager.getBestProvider	N/A
*WRITE_SETTINGS	*LocationManager.getAllProviders	N/A
*WRITE_SMS	*LocationManager.clearTestProviderStatus	N/A