Advanced Sciences and Technologies for Security Applications

Reza Montasari
Hamid Jahankhani
Richard Hill
Simon Parkinson *Editors*

Digital Forensic Investigation of Internet of Things (IoT) Devices

Springer

Advanced Sciences and Technologies for Security Applications

Indexed by SCOPUS

The series Advanced Sciences and Technologies for Security Applications comprises interdisciplinary research covering the theory, foundations and domain-specific topics pertaining to security. Publications within the series are peer-reviewed monographs and edited works in the areas of:

- biological and chemical threat recognition and detection (e.g., biosensors, aerosols, forensics)
- crisis and disaster management
- terrorism
- cyber security and secure information systems (e.g., encryption, optical and photonic systems)
- traditional and non-traditional security
- energy, food and resource security
- economic security and securitization (including associated infrastructures)
- transnational crime
- human security and health security
- social, political and psychological aspects of security
- recognition and identification (e.g., optical imaging, biometrics, authentication and verification)
- smart surveillance systems
- applications of theoretical frameworks and methodologies (e.g., grounded theory, complexity, network sciences, modelling and simulation)

Together, the high-quality contributions to this series provide a cross-disciplinary overview of forefront research endeavours aiming to make the world a safer place.

The editors encourage prospective authors to correspond with them in advance of submitting a manuscript. Submission of manuscripts should be made to the Editor-in-Chief or one of the Editors.

More information about this series at http://www.springer.com/series/5540

Reza Montasari · Hamid Jahankhani · Richard Hill · Simon Parkinson
Editors

Digital Forensic Investigation of Internet of Things (IoT) Devices

Springer

Editors
Reza Montasari
Hillary Rodham Clinton School of Law
Swansea University
Swansea, UK

Hamid Jahankhani
London Campus
Northumbria University
London, UK

Richard Hill
Department of Computer Science
University of Huddersfield
Huddersfield, UK

Simon Parkinson
Department of Computer Science
University of Huddersfield
Huddersfield, UK

In 2015, Antonio Mauro, PhD (info@antoniomauro.it) filed a patent named "Forensics Investigation in the Internet of Things (IoT) Devices".

ISSN 1613-5113 ISSN 2363-9466 (electronic)
Advanced Sciences and Technologies for Security Applications
ISBN 978-3-030-60424-0 ISBN 978-3-030-60425-7 (eBook)
https://doi.org/10.1007/978-3-030-60425-7

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Foreword

Watching the progressive rollout of the IOT, it would be easy to form the opinion that we really understand what we are doing and how the network is going to perform, but nothing could be farther from the truth! Reality is, the IOT is a new and evolutionary network form that presents levels of complexity and behavior that we never anticipated and have never seen before. Further, we do not have the tools or abilities to model, characterize, measure, and fully understand the outcomes of our designs and deployments. And along with almost all new systems, security is often omitted completely, or it appears as a weak engineering afterthought. In reality, the IoT is magnifying the attack surface of the planet to the benefit of cybercriminals and rogue states who now see the IOT as a new opportunity and entry window for wider incursions into the networks and facilities of organizations.

> *It is not possible to understate the rapidly growing cyber risks posed by The IoT or indeed the urgency of the address required*

It is, therefore, refreshing to find a book addressing this most important topic with detailed consideration of many of the initial IoT challenges. Primarily, it asks what happens when an IoT attack occurs or failure happens, and how do we locate the point of failure/entry to assess the potential consequences and affect repairs as quickly as possible? In short, the term "forensics" is a perfect fit for what is needed and what is detailed in this first book on the topic. To my mind, it represents a first and vital step in the documentation and development of a new branch of network science and engineering that is urgently required. As an academic, practitioner, and consultant in the field of cyber security, I found the treatment in each chapter refreshing and reassuring with the authors detailing their latest thoughts and research results. Best of all, they opened my mind to new concepts and avenues in the field and left me wanting for more. I, therefore, consider this to be "Volume 1" in the opening salvo of our battle for IoT security supremacy, and the survival of one of our most important components of Industry 4.0 and the realization of sustainable societies.

And so, it is in this context, and with this background that I commend this book to you as a provocative and foundation text in the field. Hopefully, you will find it enlightening and useful, and it might also spur even more innovation.

June 2020

Prof. Peter Cochrane
OBE
University of Suffolk
Ipswich, UK

Contents

Emulation Versus Instrumentation for Android Malware Detection 1
Anukriti Sinha, Fabio Di Troia, Philip Heller, and Mark Stamp

Towards a Generic Approach of Quantifying Evidence Volatility in Resource Constrained Devices 21
Jens-Petter Sandvik, Katrin Franke, and André Årnes

Application of Artificial Intelligence and Machine Learning in Producing Actionable Cyber Threat Intelligence 47
Reza Montasari, Fiona Carroll, Stuart Macdonald, Hamid Jahankhani, Amin Hosseinian-Far, and Alireza Daneshkhah

Drone Forensics: The Impact and Challenges 65
S. Atkinson, G. Carr, C. Shaw, and S. Zargari

Intrusion Detection and CAN Vehicle Networks 125
Ashraf Saber, Fabio Di Troia, and Mark Stamp

Cloud Computing Security: Hardware-Based Attacks and Countermeasures 155
Reza Montasari, Alireza Daneshkhah, Hamid Jahankhani, and Amin Hosseinian-Far

Aspects of Biometric Security in Internet of Things Devices 169
Bobby L. Tait

Evaluating Multi-layer Security Resistance to Adversarial Hacking Attacks on Industrial Internet of Things Devices 187
Hussain Al-Aqrabi and Richard Hill

Establishing Trustworthy Relationships in Multiparty Industrial Internet of Things Applications 205
Oghenefejiro Bello, Hussain Al-Aqrabi, and Richard Hill

IoT Forensics: An Overview of the Current Issues and Challenges 223
T. Janarthanan, M. Bagheri, and S. Zargari

Making the Internet of Things Sustainable: An Evidence Based Practical Approach in Finding Solutions for yet to Be Discussed Challenges in the Internet of Things 255
Benjamin Newman and Ameer Al-Nemrat

Emulation Versus Instrumentation for Android Malware Detection

Anukriti Sinha, Fabio Di Troia, Philip Heller, and Mark Stamp

Abstract In resource constrained devices, malware detection is typically based on offline analysis using emulation. An alternative to such emulation is malware analysis based on code that is executed on an actual device. In this research, we collect features from a corpus of Android malware using both emulation and on-phone instrumentation. We train machine learning models using the emulator-based features and we train models on features collected via instrumentation, and we compare the results obtained in these two cases. We obtain strong detection and classification results, and our results improve slightly on previous work. Consistent with previous work, we find that emulation fails for a significant percentage of malware applications. However, we also find that emulation fails to extract useful features from an even larger percentage of benign applications. We show that for applications that are amenable to emulation, malware detection and classification rates based on emulation are consistently within 1% of those obtained using more intrusive and costly on-phone analysis. We also show that emulation failures are easily explainable and appear to have little to do with malware writers employing anti-emulation techniques, contrary to claims made in previous research. Among other contributions, this work points to a lack of sophistication in Android malware.

A. Sinha (✉) · F. Di Troia · P. Heller · M. Stamp
San Jose State University, San Jose, CA, USA
e-mail: anukriti.sinha@sjsu.edu

F. Di Troia
e-mail: fabioditroia@msn.com

P. Heller
e-mail: philip.heller@sjsu.edu

M. Stamp
e-mail: mark.stamp@sjsu.edu

R. Montasari et al. (eds.), *Digital Forensic Investigation of Internet of Things (IoT) Devices*, Advanced Sciences and Technologies for Security Applications,
https://doi.org/10.1007/978-3-030-60425-7_1

1 Introduction

In 2007, Google launched a mobile operating system (OS) known as Android, which is based on the Linux kernel and other open source software. Android is used primarily on touchscreen devices such as tablets and smartphones. Google distributes Android as an open source platform, which has encouraged the use of smartphones as a computing platform [29]. Android currently dominates the mobile OS market, with more than a billion Android devices having been sold, and more than 65 billion applications (apps) having been downloaded from the Google Play Store. Android devices account for more than 80% of the overall mobile OS market [13].

The prominence of Android has not escaped the attention of malware writers. According to McAfee, more than 3,000,000 Android malware apps were detected in 2017, representing a 70% increase from 2016. Also, in 2017 alone, more than 700,000 malicious apps were removed from the Google PlayStore [20].

In resource constrained devices, such as Android smartphones, malware detection is typically conducted offline, based on emulation. The objective of this research is to explore the effectiveness of malware detection and classification using dynamic features extracted via emulation, as compared to extracting such features via instrumentation (i.e., on-phone analysis). We classify Android apps using a wide variety of machine learning techniques based on these emulator-extracted and "real" features. We find that emulation fails for a significant percentage of apps and that, surprisingly, the failure rate is higher for benign apps than malicious apps. In contrast to claims that appear in the research literature [25], we find scant evidence that such failures are due to anti-emulation techniques being employed by sophisticated Android malware. Instead, the evidence indicates that Android malware writers fail to take advantage of relatively simple techniques that could serve to make the detection problem considerably more challenging [6, 22, 34].

We note that our analysis technique closely follows that in [2]. However, we go beyond the work in [2] in that we consider additional machine learning techniques, we tune the parameters, and in addition to the detection problem, we also consider the classification problem. Furthermore, we show that a simple ensemble technique can provide essentially ideal separation for the malware detection problem. Finally, with respect to the sophistication of Android malware, we draw diametrically opposed conclusions, as compared to previous work such as [25].

The remainder of this paper is organized as follows. Section 2 provides an overview of various feature analysis methods that have previously been used to successfully detect Android malware, along with an overview of selected examples of Android malware research. In Sect. 3, we discuss the methodology used in our experiments. Section 4 gives our experimental results, along with some discussion of the implications of these results. Finally, Sect. 5 concludes the paper and outlines possible directions for future work.

2 Background

In this section, we first discuss the relative advantages and disadvantages of static and dynamic features for malware analysis. Then we briefly consider the potential weaknesses of emulator-based detection for Android malware.

2.1 Static and Dynamic Features

Malware detection can be based on static or dynamic features. Features are said to be static if they are collected without executing (or emulating) the code. On the other hand, dynamic features require code execution or emulation. Examples of popular static features include byte n-grams and mnemonic opcodes, while useful dynamic features include opcodes and application programming interface (API) calls that occur when an app executes. In general, static features can be collected more efficiently than dynamic features. The relative advantage of dynamic features is that detection techniques based on such features are often more robust with respect to common obfuscation techniques [7]. In the Android malware literature, both static and dynamic features have been extensively studied [16].

An Android app consists of a package bundled as an Android Package file, which has the file extension `apk`. Among other things, an `apk` file contains a manifest (`AndroidManifest.xml`), class files (`classes.dex`), and external libraries. Figure 1 lists the components of an `apk` bundle, while Fig. 2 gives an example of a typical manifest file.

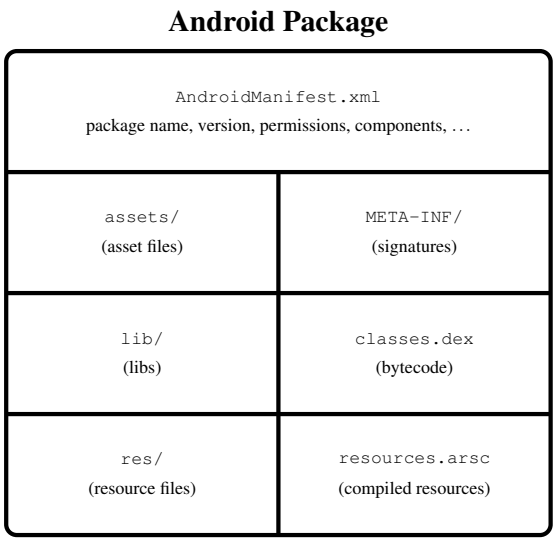

Fig. 1 The parts of an `apk` bundle

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.novaapps.findevents" android:versionCode="1"
    android:versionName="1.0"
    android:installLocation="preferExternal">
    <uses-sdk android:minSdkVersion="4" />
    <supports-screens
        android:largeScreens="true"
        android:normalScreens="true"
        android:smallScreens="true"
        android:resizeable="true"
        android:anyDensity="true" />

<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_LOCATION_EXTRA_COMMANDS" />
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />

    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name=".FindEventsGADroidActivity"
            android:label="@string/app_name" android:configChanges="orientation|keyboardHidden">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <activity android:name="com.phonegap.DroidGap" android:label="@string/app_name"
            android:configChanges="orientation|keyboardHidden">
            <intent-filter>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Fig. 2 Sample `AndroidManifest.xml` file

Many useful static features can be extracted directly from the manifest file. A considerable amount of malware research has focused on static Android features such as permissions (functionality requested by the app). For example, it has been found that number of permissions requested is a surprisingly strong diagnostic [17], with malicious apps requesting more permissions, on average, than benign apps. However, notwithstanding the relative ease and computational efficiency of static analysis, this approach has a significant drawback, as it is relatively easy for malware writers to evade static detection by obfuscating their code. Obfuscation tools are readily available; for example, ProGuard can change data pathnames, variable names, and function names [23].

Dynamic analysis consists of extracting features while code is executing, either on the device for which the code is intended or on an emulator [18]. Some popular dynamic Android features include kernel processes, API calls, and information related to dynamic loading. Dynamic techniques often deal with analyzing internal system calls made by an application at runtime [18]. Previous work has demonstrated the advantage of dynamic features over static features for malware detection [7]. However, the increased efficiency of static feature extraction makes static analysis preferable in cases where it can achieve results that are comparable to dynamic analysis.

To analyze features—static, dynamic, or some combination thereof—researchers can employ a wide variety of machine learning techniques. Examples of such machine learning techniques include k-nearest neighbors, hidden Markov models, principal

component analysis, support vector machines, clustering, and deep neural networks, among others [30].

From the malware writer's perspective, it is desirable to make a malicious app appear benign under any anticipated analysis. A variety of obfuscation techniques (e.g.. dead code insertion and code substitution) are available to disguise malware and are generally most effective during static analysis. A variety of anti-emulation techniques are available for evading detection by dynamic feature extraction under emulation. These are best understood in the context of the following discussion of emulation.

2.2 Emulation

Android malware can access sensitive information such as call history, text and contacts, and can tamper with phone settings. To do this, malicious apps often try to read the background environment via API calls. Examining the result of selected API calls can enable a malicious app to identify the environment on which the code is executing and thereby determine how best to attack the device [2]. Emulators are not entirely faithful to real phone APIs, and malicious apps can use these discrepancies to detect when they are being executed in an emulated environment and therefore should restrict suspicious behavior. An example of an API that can be used to detect emulators is the Telephony Manager API, that is,

```
TelephonyManager.getDeviceId()
```

A call to this API typically returns `000000000000000` when an emulator is executing the code. A real physical device, on the other hand, would not return 0 as the device identifier. This is one of the emulator detection methods that is used by the Pincer family of Android malware [34]. Emulator detection is a significant challenge to security analysis, because most emulators use open source hypervisors such as QEMU, which have detectable identifying functionality [15]. It has been claimed that the Morpheus malware app employs more than 10,000 heuristics to classify its runtime environment [3].

To deal with issues such as these, researchers have attempted to develop improved emulators. Several dynamic analysis tools such as TaintDroid [9], DroidBox [9], CopperDroid [32], Andrubis [19, 35], and AppsPlayground [26] have been developed. In addition, online tools are available for Android application analysis, including SandDroid [27], TraceDroid [33], and NVISO ApkScan [21]. However, these dynamic approaches still rely on emulators or virtualized environments which malware can detect by careful analysis [21].

Since it is possible for Android malware to detect an emulated environment, we might assume that malware would check for emulation and behave benignly when an emulator is detected. Indeed, it has been claimed that such is the case for most Android malware [2]. However, our results indicate that the Android malware datasets used in

our experiments are not, in general, using advanced emulation avoidance techniques to any greater degree than benign apps. This observation is based, in part, on the fact that we find that benign apps fail to run in our emulation environment at a higher rate than malware. In addition, we find that these failures are easily explained by the limitations of the emulation environment, rather than advanced anti-emulation strategies.

2.3 Selected Android Malware Research

The authors of the paper [10] study packed Android malware. These authors show that in the time period from 2010 to 2015, about 13% of the Android malware that they consider was packed, and that sophisticated Android malware samples often use (and abuse) custom packers. Similar to code encryption, packing is a well-known technique for defeating signature-based and some other static detection techniques [4]. However, in this paper, we only consider dynamic analysis, which should be unaffected by code packing.

The work [5] considers the problem of detecting privacy leak caused by Android malware. The authors employ a differential analysis technique, here they vary certain key parameters and look for changes in network activity that are evidence of private data leaking. The authors show that their technique is practical and effective. Additional research on the privacy leak problem can be found in [31], where the authors develop and analyze an information flow analysis tool, TAINTART, which can be viewed as an improved version of TaintDroid [37]. Such privacy leakage and information flow work is relevant to the problem consider in this paper, and it serves to illustrate ways that, for example, features could be collected in an Android environment.

The research presented in [12] considers the interesting and challenging problem of detecting Android malware that contains a "logic bomb," which the authors define to be malicious code that only executes under some narrow circumstance. Such code might be used, for example, in an attack that is carefully targeted at a specific user or other entity, and seems to be relatively common in malware developed by nation states. This paper is focused on a narrow and apparently rare class of Android malware, whereas our research considers the general Android malware detection problem.

The main insights in the paper [11] is that Android intents are a stronger feature than permissions. An Android intent is a messaging object that can be used to request an action from another app component [14]. While permissions have been extensively studied in the literature, intents have received much less attention. The authors of [11] also consider a combination of the two feature types—intents and permissions—and show that this yields improved results, as compared to using intents only.

The authors of [25] have developed a tool to extract runtime features, from obfuscated Android malware. For example, encrypted SMS numbers cannot be detected

via static analysis and malware that can detect an emulation environment could also hide such data at runtime.

Finally, we note that many research papers claim that it is common for Android malware apps to employ emulation-detection techniques to hide features, while many other research papers implicitly assume that such is the case. Indeed, this assumption is the impetus for considerable research in the Android malware domain. For example, in [25], it is stated that "many malicious applications" use emulation-detection techniques, but no evidence is provided as to the percentage of such applications that actually occur in their malware dataset. Furthermore, the papers cited in [25] as evidence of the supposed widespread use of such detection-avoidance techniques, namely [6, 22, 34], do not provide such numbers either, and instead simply show that it is possible (and, in fact, relatively easy) to implement such feature-hiding capabilities. We return to this issue in Sect. 5.

3 Methodology

This section describes the process we followed to dynamically extract features from Android apps. We extract such features from both benign and malicious apps, using both emulation and on-phone instrumentation. But first, we briefly discuss the datasets used in our experiments before providing details on the feature extraction process.

3.1 Datasets

AMGP dataset This dataset is part of the Android Malware Genome Project [38], and it has been used in numerous research papers, including [2]. Of the 2444 apps in the dataset, half are malicious apps from 49 different families, with the remainder being benign apps from McAfee Labs [2]. We use this dataset for binary classification experiments, where we classify samples as either malware or benign.

Drebin dataset We also experiment with 3206 samples from the seven malware families in the Drebin dataset [8]. The list of families and the number of samples from each are given in Table 1. This dataset was used in experiments where we attempted to classify samples into their respective families, as opposed to binary classification (i.e., malware and benign) experiments

Table 1 Drebin data

Family	Apps
FakeInstaller	925
DroidKungfu	667
Plankton	625
Opfake	613
Iconosys	152
Fakedoc	132
Geinimi	92
Total	3206

3.2 Feature Extraction

Feature extraction is a critical aspect of this research, as our approach is based on comparing results from various machine learning techniques, using features collected via emulator versus features collected directly from a phone-based environment. Therefore, feature extraction was performed for both environments, as described below.

Phone environment The Android smartphone used for data collection was configured as follows: Android 5.0 Lollipop, 1.3 GHz CPU, 16 GB internal memory, and 32 GB of external SD card storage. The phone contained a SIM card with activated service to enable 3G data use and outgoing calls. This configuration is consistent with that used in [2]. As discussed in [2], USB 2.0 or 3.0 was used along with the Linux VM so as to avoid the timeout that would result from a USB 1.0 connection with files larger than 1MB.

Emulation environment A Santoku Linux VirtualBox was used to emulate an Android device. The environment was configured as follows: 8 GB of external SD card memory, 2 MB of memory, 4.1.2 Jelly Bean (API level 16, Android version). To more accurately simulate the workings of a real phone, the emulator was enhanced with contact numbers, images, pdf files, and text files. The default IMEI, IMSI, SIM serial number, and phone numbers were altered. After each application was executed, the emulator was re-initialized to ensure the removal of third party apps. This emulation process is consistent with that used in [2].

DynaLog is a dynamic framework that accepts a large number of apps as input, launches them serially in the emulator environment, creates logs of dynamic features, and extract these features for future processing [1]. At the core of DynaLog is the MonkeyRunner API that is able to stimulate apps with random events that are typical of user interactions (e.g.. pressing, swiping, and touching the screen). These simulated actions are designed to stimulate a significant fraction of code functionality.

To extract dynamic features from the phone, we call DynaLog using a Python based tool, as described in [1]. Each app was executed for 15 min during which time we logged and collect dynamic features from the phone, as well as from an emulator

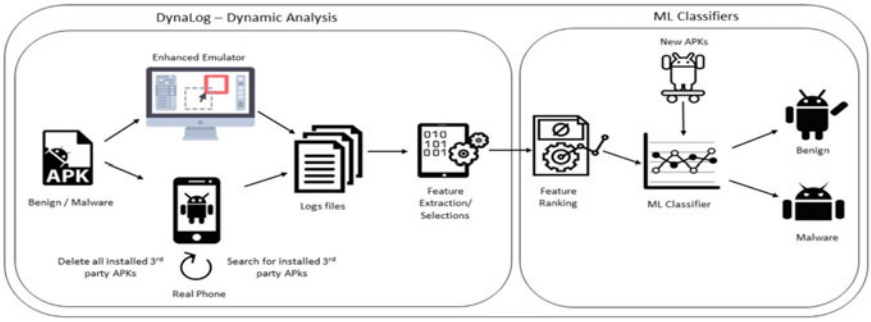

Fig. 3 DynaLog [2]

running the same apps with the same input events. Figure 3 illustrates the use of DynaLog [2].

The data collected from the phone and emulator was saved in files in the `arff` format suitable for feature vector input to machine learning platforms. The 178 features form these vectors were loaded into Weka [36]. The features were then ranked based on information gain (`InfoGain` in Weka) and the top 100 features from each analysis environment (phone and emulation) were then used to test and train the machine learning algorithms considered in this paper.

3.3 Machine Learning Models

In this section, we briefly discuss each of the nine machine learning techniques used in this research. These nine algorithms cover a broad range of techniques, ranging from relatively simple statistical scores to advanced neural networks.

Support Vector Machine A support vector machine (SVM) represents data as points in a high-dimensional space, and computes a hyperplane or manifold that separates points of different classes. The multiclass version of an SVM is known as a support vector classifier (SVC).

Naïve Bayes This approach uses Bayes' theorem to compute probabilities of data points belonging to classes. To simplify computation, features are "naïvely" assumed to be independent of each other even when they are actually dependent.

Simple Logistic Simple logistic is an ensemble learning algorithm that uses multiple simple regression functions to model the training data, computing weights that maximize the log-likelihood of the logistic regression function.

Multilayer Perceptron A multilayer perceptron (MLP) is a feedforward neural network that includes an input layer, an output layer, and one or more hidden layers. MLPs are trained by backpropagation with gradient descent to minimize errors.

IBk This is the Weka implementation of the k-nearest neighbors algorithm, using a Euclidean distance metric to define "nearest." Given an integer k, the algorithm classifies a point in feature space by considering its k nearest classified neighbors.

Partial Decision Trees A partial decision tree (PART) is a simple decision tree that contains branches to undefined sub-trees. In order to develop a partial decision tree, construction and pruning operations are used, with the goal of finding a sub-tree that cannot be further simplified.

J48 Decision Tree An implementation of the C4.5 decision tree algorithm, J48 repeatedly splits on the remaining feature with highest information gain.

Random Forest The random forest (RF) technique relies on a "forest" of decision trees. That is, multiple decision trees are trained, and a majority vote of the trees is used for classification. The RF algorithm uses bagging, whereby subsets of features and samples are selected to construct the component trees. Bagging enables a random forest to greatly reduce the overfitting problem that is inherent in elementary decision trees.

AdaBoost Boosting is a general machine learning technique that can build a strong classifier from a number of weak classifiers. AdaBoost uses a simple adaptive strategy to build such a classifier. The implementation of AdaBoost that we employ is based on decision tree classifiers.

3.4 Evaluation Metrics

From the point of view of this analysis, a positive classification is an identification as malware. We tabulated true/false positive/negative rates for all analyses. Sensitivity and recall are terms that are equivalent to true positive rate. Precision is the ratio of true positives to the number of samples that are classified as positives. Thus, in our binary classification experiments, precision tells us the fraction of samples classified as malware that are actually malware. The primary metric we use in this paper is the F-measure, which is defined as

$$\text{F-measure} = \frac{2 \times \text{precision} \times \text{recall}}{\text{recall} + \text{precision}}$$

By combining both precision and recall into a single statistic, the F-measure provides a useful single value for comparing machine learning approaches.

4 Experiments and Results

This section presents the results of two broad classes of experiments. Our first category of experiments deals with evaluating the effectiveness of Android malware detection based on features extracted via emulation, as compared to features extracted

Table 2 AMGP dataset feature extraction success

Type	Emulator		Phone	
	Number	Percentage	Number	Percentage
Malware	956	78.23	1211	99.09
Benign	807	66.03	1119	91.57
Total	1763	72.13	2330	95.33

Table 3 Drebin dataset feature extraction success

	Emulator	Phone
Number	2598	3201
Percentage	81.03	99.84

directly from a phone. In these experiments, we use the same dataset and feature extraction tools as in [2]. Moreover, we consider additional machine learning techniques, we tune the parameters of the machine learning algorithms,[1] we consider a multi-sensor solution, and we ultimately draw somewhat different conclusions based on our results. We refer to this first set of experiments as malware detection experiments.

Our second set of experiments involves classifying malware samples into families. Again, we consider a variety of machine learning algorithms and we compare the results obtained when using emulator and phone-based features. We refer to this second set of experiments as malware classification experiments.

Before presenting these experimental results, we first discuss the data collection phase in some detail. This is an important issue, since we were not able to extract features from all apps using the automated approach considered here.

4.1 Emulation Versus Instrumentation

Table 2 gives the percentage of apps from the AMGP dataset that we were able to analyze using emulation, as well as the percentage of apps that we could evaluate using on-phone analysis. Table 3 gives analogous results for the Drebin dataset. Recall that the AMGP dataset is evenly split between malware and benign apps, with 1222 in each category; the Drebin dataset contains 3206 malware apps, with the breakdown by family given in Table 1.

Tables 2 and 3 show that nearly 20% fewer malicious Android apps allow for feature extraction using emulation, as compared to the on-phone environment, and this is consistent across both datasets. This has led some researchers to conclude

[1] Based on our experiments, it appears that the authors of [2] consistently used the Weka default settings for their machine learning experiments.

Table 4 Features extracted only from phone environment (AMGP dataset)

Feature	Phone	Emulator
`System;loadLibrary`	212	0
`URLConnection;connect`	15	0
`Context;unbindService`	4	0
`Service;onCreate`	3	0
`BATTERY_LOW`	1	0
`SmsManager;sendTextMessage`	3	0

that anti-emulation techniques must be widely used in Android malware. If such is the case, it is not clear why benign apps would use anti-emulation techniques at an even higher rate than malicious apps—compare the benign and malware results in Table 2. This raises questions as to whether the results for malicious apps are really due to anti-emulation techniques, or whether there might be another explanation.

A more plausible reason why we are able to automatically extract features from more apps using on-phone instrumentation is simply because more APIs can be executed on a phone environment. This is especially an issue for apps that make API calls related to network activity or read incoming and outgoing call activity. Whether such apps are benign or malicious, the phone is able to provide such capabilities and thereby log the relevant API activity, while emulators are not sufficiently sophisticated to simulate all necessary APIs. Manual analysis of a number of apps that fail under emulation reveals that network and call-related issues are indeed responsible for emulation failures for both malicious and benign apps.

Table 4 lists the features that were extracted exclusively from the phone but not by the emulator. For example, the `System.loadLibrary` feature is the API call associated with native code; it is probably not logged under emulation because the emulator does not support native code [2]. The phone based analysis shows a much higher effectiveness in extracting features for analysis; this is clearly an essential benefit for machine learning classification.

4.2 Binary Classification Experiments

In this section, we give the results for binary classification experiments using the AMGP dataset. We consider each of the nine machine learning techniques discussed in Sect. 3.3, and compare the results for features extracted via emulation against results for features extracted via on-phone instrumentation. All experiments were performed using Weka with 10-fold cross validation. The models were fine-tuned over various input parameters, with the following list giving some of the important settings.

Table 5 Results for emulator based features (AMGP dataset)

Model	TPR	FPR	TNR	FNR	F-measure
Simple logistic	0.902	0.097	0.903	0.098	0.901
Naïve Bayes	0.599	0.098	0.902	0.401	0.734
SVM	0.914	0.094	0.906	0.086	0.908
PART	0.902	0.099	0.901	0.098	0.899
J48	0.892	0.116	0.884	0.108	0.886
RF	0.916	0.063	0.937	0.084	0.928
MLP	0.941	0.087	0.913	0.059	0.926
IBk	0.899	0.096	0.904	0.101	0.903
AdaBoost	0.901	0.101	0.899	0.099	0.900

Simple Logistic The ridge estimator for regularization is used to reduce the size of coefficients. The model is trained until it converges.

Naïve Bayes Default values are used for the kernel and for discretization.

Support Vector Machine The complexity parameter C is set to 1.0 and a polynomial kernel is used.

Decision Trees We experimented with various depths for the trees (the `maxDepth` parameter in Weka) and the best accuracy was obtained with a depth of 50. The `noPruning` option was set to `False`.

Random Forest The model yielded the best accuracy with 100 trees and this is what we use in all experiments reported here.

Multilayer Perceptron The number of hidden layers is chosen to be 3.

IBk We use the Euclidean distance with 10 neighbors.

AdaBoost The classifier we use is the decision stump algorithm.

Using features collected from the emulator, we obtain the results in Table 5. From these results, we see that the best accuracy is achieved by a random forest with 100 trees, while an MLP yields a similar F-measure.

For our next set of experiments, we repeat the above analyses, but using features extracted via on-phone instrumentation, with all algorithms parameterized exactly as in the emulation case. Results for these experiments are summarized in Table 6. As with the emulation-based results, the random forest and MLP again perform the best.

We performed another set of experiments on the AMGP dataset using only those apps that successfully executed in both the emulator and on-phone environments. For these apps, the results of testing and training the various machine learning models based on features extracted from the emulator are given in Table 7.

The results in Table 7 again show that random forest yielded the best results. Furthermore, the random forest experiments in Table 7 yielded nearly identical results to those in Table 5. However, for the other techniques, the results are generally

Table 6 Results for phone based features (AMGP dataset)

Model	TPR	FPR	TNR	FNR	F-measure
Simple Logistic	0.923	0.081	0.919	0.077	0.921
Naïve Bayes	0.634	0.119	0.881	0.366	0.748
SVM	0.918	0.090	0.910	0.082	0.914
PART	0.907	0.098	0.902	0.093	0.905
J48	0.929	0.101	0.899	0.071	0.916
RF	0.942	0.074	0.926	0.058	0.934
MLP	0.924	0.082	0.918	0.076	0.925
IBk	0.906	0.086	0.914	0.094	0.910
AdaBoost	0.908	0.087	0.913	0.092	0.906

Table 7 Apps executed in both environments (AMGP data and emulator features)

Model	TPR	FPR	TNR	FNR	F-measure
Simple Logistic	0.887	0.104	0.896	0.113	0.891
Naive Bayes	0.542	0.169	0.831	0.458	0.663
SVM	0.896	0.116	0.884	0.104	0.889
PART	0.896	0.116	0.884	0.104	0.892
J48	0.874	0.088	0.912	0.126	0.894
RF	0.919	0.066	0.934	0.081	0.927
MLP	0.898	0.096	0.904	0.102	0.902
IBk	0.904	0.090	0.910	0.096	0.907
AdaBoost	0.901	0.093	0.907	0.099	0.902

slightly lower than either the exclusively emulator-based or instrumentation-based experiments considered above.

In order to assess the value of analysis with multiple machine learning models, error rates were considered as a function of the number of models. In Table 8 (a), we give results for false negatives (FN), for both the emulator and on-phone features. The row labeled with n in the table gives the number of malware apps that were misclassified as benign by n or more of the nine machine learning techniques considered, based on emulator features (middle column) or on-phone features (last column). Table 8 (b) gives the analogous results for false positives. These results are summarized in the form of line graphs in Fig. 4.

Suppose that we base our classification on a majority vote of the nine machine learning models considered above. Then the numbers in Table 8 (a) and (b) imply that when using the emulator features, we would have only 7 false negatives and 3 false positives, while the corresponding numbers for the on-phone features is 3 false negatives and 0 false positives. The corresponding accuracies and F-measures are

Table 8 Classification errors and machine learning models (AMGP dataset)

Number of models	Features	
	Emulator	Phone
(a) *False negatives*		
1	104	91
2	72	64
3	36	28
4	19	11
5	7	3
6	2	0
7	0	0
8	0	0
9	0	0
(b) *False positives*		
1	73	62
2	46	30
3	21	16
4	9	4
5	3	0
6	0	0
7	0	0
8	0	0
9	0	0

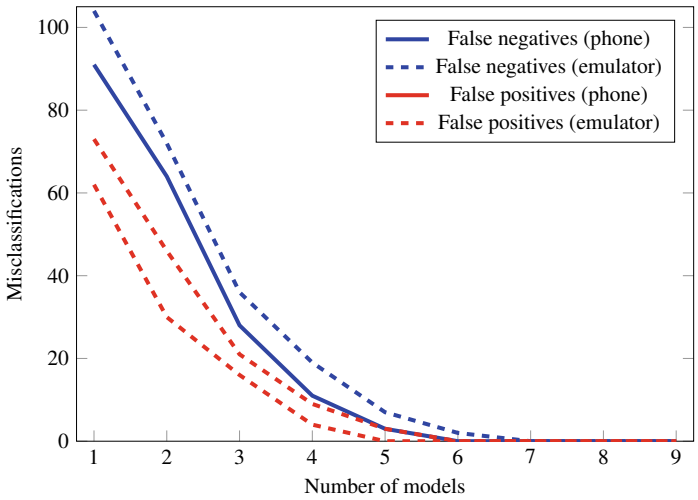

Fig. 4 Misclassifications as a function of the number of models

Table 9 Majority vote of machine learning models (AMGP dataset)

Features	Accuracy	F-measure
Emulator	0.9960	0.9959
Phone	0.9988	0.9988

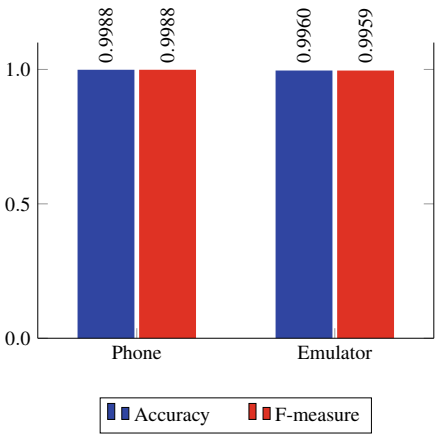

Fig. 5 Majority vote of models

given in Table 9 and in the form of bar graphs in Fig. 5. These results—which are virtually ideal—are far stronger than any of the individual models, and indicate the potential strength of a multi-sensor approach. More sophisticated techniques of combining the output of multiple machine learning models could potentially yield equally strong results with fewer models. For example, in the malware domain, SVMs have been used to combine multiple scores into a single machine learning model [28] and boosting techniques can produce a strong combined classifier [24].

4.3 Multiclass Experiments

In this section we give multiclass results based on a support vector classifier (SVC), which is the mulitclass version of an support vector machine (SVM). For these experiments, we employ the Drebin dataset and we use a linear kernel in all cases. As in the binary classification experiments above, the goal is to compare the performance of models trained on features that have been extracted using on-phone instrumentation with models trained on features extracted via emulation. We expect the multi-family classification problem to be inherently more challenging than the binary classification (malware versus benign) problem due to the larger number of classes.

Table 10 shows the results for our multiclass experiments, with Fig. 6 giving these same results in the form of line graphs. This table and figure include results for both feature extraction environments (emulation and on-phone instrumentation). Note that

Table 10 Family classification results (Drebin dataset)

Families	Combinations	Emulator	Phone
2	21	0.9278	0.9364
3	35	0.9182	0.9276
4	35	0.9113	0.9202
5	21	0.9079	0.9184
6	7	0.8982	0.9064
7	1	0.8890	0.8997

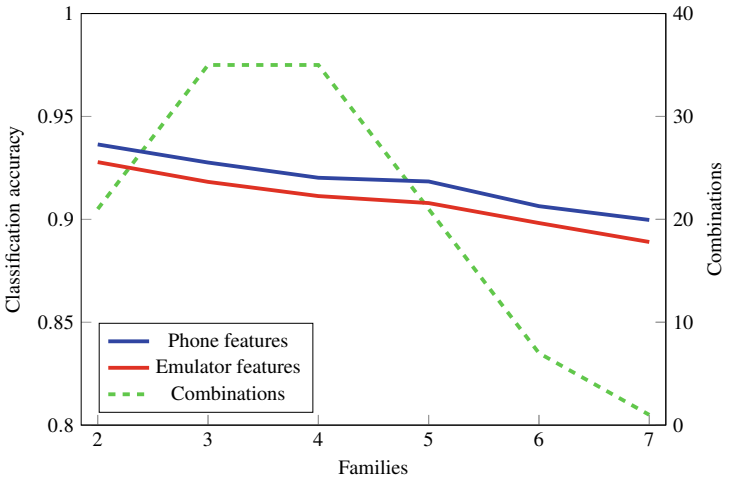

Fig. 6 Family classifications

there are seven families in the Drebin dataset (see Table 1), and we have conducted experiments with each of the 127 nontrivial combinations of these families. The accuracy reported in Table 10 and Fig. 6 for k families is the average of all $\binom{7}{k}$ possible combinations of k families. From these results we see that the on-phone features yield consistently better results than the emulation features, but—as with the binary classification experiments discussed above—the differences are slight. It is interesting that the classification accuracies are so high, which seems to indicate that the families in this dataset may differ substantially from one another.

5 Conclusion and Future Work

In this research, we have considered Android malware detection and classification. Our primary focus was to compare the effectiveness of features extracted on-phone with features extracted using emulation and to consider the implications of these

results. In our binary classification experiments we considered nine machine learning techniques (support vector machines, random forest, naïve Bayes, multilayer perceptron, simple logistic, J48 decision tree, PART, IBk, and AdaBoost). We used support vector machines in our classification experiments.

In all cases, we obtained strong results as measured by the F-measure statistic. Although the on-phone features performed marginally better than the emulation features, we conclude that the additional overhead of on-phone analysis is unlikely to be worthwhile in most situations. That is, the incremental reduction in error rates is unlikely to be cost-effective.

A simple majority vote of our nine classifiers yielded essentially perfect detection and F-score results, as given in Table 9. These results exceed those found in previous work, such as [2].

Our results also call into question the oft-stated claim that Android malware frequently uses anti-emulation techniques. Instead, we believe that these experiments offer evidence that Android malware is actually much less sophisticated than is sometimes claimed. In fact, this is easily confirmed by a manual analysis of apps—malware and benign—that fail in the emulation environment. We find that such apps fail simply due to the inability of the emulator to handle call, networking, and similar APIs.

Future work could include a similar analysis on larger and more recent Android malware datasets. While it is not the case that anti-emulation was effectively used by the malware in our datasets, it would not be difficult for a moderately skilled malware writer to generate apps that would be much more challenging to detect. Work involving a more recent dataset would be a way to determine whether Android malware writers have started taking advantage of such techniques.

References

1. Alzaylaee MK, Yerima SY, Sezer S (2016) DynaLog: an automated dynamic analysis framework for characterizing Android applications. In: 2016 international conference on cyber security and protection of digital services, Cyber Security 2016, pp 1–8. arXiv:1607.08166
2. Alzaylaee MK, Yerima SY, Sezer S (2017) EMULATOR vs REAL PHONE: Android malware detection using machine learning. In: Proceedings of the 3rd ACM on international workshop on security and privacy analytics, IWSPA '17, pp 65–72
3. Amos B, Turner HA, White J (2013) Applying machine learning classifiers to dynamic Android malware detection at scale. In: 9th international wireless communications and mobile computing conference, IWCMC 2013, pp 1666–1671
4. Aycock J (2006) Computer viruses and malware. Advances in information security. Springer US
5. Continella A, Fratantonio Y, Lindorfer M, Puccetti A, Zand A, Krügel C, Vigna G (2017) Obfuscation-resilient privacy leak detection for mobile apps through differential analysis. In: 24th annual network and distributed system security symposium, NDSS, 2017. http://www.s3.eurecom.fr/~yanick/publications/2017_ndss_agrigento.pdf
6. Coogan K, Debray S, Kaochar T, Townsend G (2009) Automatic static unpacking of malware binaries. In: 16th working conference on reverse engineering, WCRE 2009, pp 167–176

7. Damodaran A, Di Troia F, Visaggio CA, Austin TH, Stamp M (2017) A comparison of static, dynamic, and hybrid analysis for malware detection. J Comput Virol Hacking Tech 13(1):1–12
8. The Drebin dataset. https://www.sec.cs.tu-bs.de/~danarp/drebin/
9. DroidBox Google archive. https://code.google.com/archive/p/droidbox/
10. Duan Y, Zhang M, Bhaskar AV, Yin H, Pan X, Li T, Wang X, Wang X (2018) Things you may not know about Android (un) packers: a systematic study based on whole-system emulation. In: 25th annual network and distributed system security symposium, NDSS, pp 18–21. https://www.informatics.indiana.edu/xw7/papers/ndss18-paper296.pdf
11. Feizollah A, Anuar NB, Salleh R, Suarez-Tangil G, Furnell S (2017) AndroDialysis: analysis of Android intent effectiveness in malware detection. Comput Secur 65:121–134. http://www0.cs.ucl.ac.uk/staff/G.SuarezdeTangil/papers/2017cosec-androdialysis.pdf
12. Fratantonio Y, Bianchi A, Robertson W, Kirda E, Kruegel C, Vigna G (2016) Triggerscope: towards detecting logic bombs in Android applications. In: 2016 IEEE symposium on security and privacy, SP 2016, pp 377–396. https://sites.cs.ucsb.edu/~vigna/publications/2016_SP_Triggerscope.pdf
13. Global smartphone shipments by OS. https://www.statista.com/statistics/263437/global-smartphone-sales-to-end-users-since-2007/
14. Intents and intent filters: Android developers guide. https://developer.android.com/guide/components/intents-filters
15. Jing Y, Zhao Z, Ahn G-J, Hu H (2014) Morpheus: automatically generating heuristics to detect Android emulators. In: Proceedings of the 30th annual computer security applications conference, ACSAC '14, pp 216–225,
16. Kang H, Jang J, Mohaisen A (2015) Kim HK (2015) Detecting and classifying Android malware using static analysis along with creator information. Int J Distrib Sens Netw 7(1–7):9
17. Kapratwar A, Di Troia F, Stamp M (2017) Static and dynamic analysis of Android malware. In: Mori P, Furnell S, Camp O (eds) Proceedings of the 3rd international conference on information systems security and privacy, ICISSP 2017, Porto, Portugal. SciTePress, pp 653–662, 19–21 Feb 2017
18. Lindorfer M, Neugschwandtner M, Platzer C (2015) MARVIN: efficient and comprehensive mobile app classification through static and dynamic analysis. In: IEEE 39th annual computer software and applications conference, COMPSAC 2015, pp 422–433
19. Lindorfer M, Neugschwandtner M, Weichselbaum L, Fratantonio Y, van der Veen V, Platzer C (2014) Andrubis–1,000,000 apps later: a view on current Android malware behaviors. In: Proceedings of the international workshop on building analysis datasets and gathering experience returns for security, BADGERS 2014, Wroclaw, Poland, Sept 2014
20. McAfee threats report 2017. https://www.mcafee.com/us/resources/reports/rp-quarterly-threats-dec-2017.pdf
21. NVISO Apkscan. https://apkscan.nviso.be/
22. Petsas T, Voyatzis G, Athanasopoulos E, Polychronakis M, Ioannidis S (2014) Rage against the virtual machine: hindering dynamic analysis of Android malware. In: Proceedings of the seventh European workshop on system security, EuroSec '14, pp 5:1–5:6
23. Pincer Android attacks. https://www.fsecure.com/weblog/archives/00002538.html
24. Raghavan A, Di Troia F, Stamp M (2019) Hidden Markov models with random restarts versus boosting for malware detection. J Comput Virol Hacking Tech 15(2):97–107
25. Rasthofer S, Arzt S, Miltenberger M, Bodden E (2016) Harvesting runtime values in Android applications that feature anti-analysis techniques. In: 23rd annual network and distributed system security symposium, NDSS, 2016. https://www.bodden.de/pubs/ssme16harvesting.pdf
26. Rastogi V, Chen Y, Enck W (2013) AppsPlayground: automatic security analysis of smartphone applications. In: Proceedings of the third ACM conference on data and application security and privacy, CODASPY '13, pp 209–220
27. SandDroid—an automatic Android application analysis system. http://sanddroid.xjtu.edu.cn/
28. Singh T, Di Troia F, Visaggio CA, Austin TH, Stamp M (2016) Support vector machines and malware detection. J Comput Virol Hacking Tech 12(4):203–212

29. Smartphone OS market share worldwide 2009–2017. https://www.statista.com/statistics/263453/global-market-share-held-by-smartphone-operating-systems
30. Stamp M (2017) Introduction to machine learning with applications in information security. Chapman and Hall/CRC, Boca Raton
31. Sun M, Wei T, Lui JC (2016) TaintART: a practical multi-level information-flow tracking system for Android runtime. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, CCS '16, pp 331–342. https://www.cse.cuhk.edu.hk/~cslui/PUBLICATION/CCS16.pdf
32. Tam K, Khan SJ, Fattori A, Cavallaro L (2015) CopperDroid: automatic reconstruction of Android malware behaviors. In: NDSS symposium, NDSS 2015, pp 8–11
33. Tracedroid. https://github.com/ligi/tracedroid
34. Vidas T, Christin N (2014) Evading Android runtime analysis via sandbox detection. In: Proceedings of the 9th ACM symposium on information, computer and communications security, ASIA CCS '14, pp 447–458
35. Weichselbaum L, Neugschwandtner M, Lindorfer M, Fratantonio Y, van der Veen V, Platzer C (2014) Andrubis: Android malware under the magnifying glass. Technical Report TR-ISECLAB-0414-001, Vienna Univeristy of Technology, 5
36. Weka 3: machine learning software in Java. https://www.cs.waikato.ac.nz/ml/weka/index.html
37. Yan L-K, Yin H (2012) DroidScope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In: USENIX security symposium, USENIX 2012, pp 569–584. http://www.cs.columbia.edu/~lierranli/coms6998-11Fall2012/papers/droidscope_usenixsec2012.pdf
38. Zhou Y, Jiang X (2012) Android malware genome project. http://www.malgenomeproject.org