An Intelligent Multi-Agent based Detection Framework for Classification of Android Malware

by

Mohammed Shahidul Alam

B.Sc., The University of Texas, Austin, 2002

M.Sc., The University of British Columbia, Vancouver, 2005

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Computer Science)

The University of British Columbia

(Vancouver)

December 2016

© Mohammed Shahidul Alam, 2016

Abstract

Smartphones play an important role in our day to day activities. Some of them include monitoring our health such as eating habits, sleep patterns and exercise schedule. The Android mobile operating system developed by Google is currently the most popular operating system for such smart devices. It is also the most vulnerable device due to its open nature of software installation, ability to dynamically load code during runtime, and lack of updates to known vulnerabilities even on popular versions of the system. Thus, securing such devices from malware that targets user privacy and monetary resources is paramount.

In this thesis, we developed a context-aware multi-agent based framework targeted towards protecting Android devices. A malware detection technique has to be context-aware due to limited battery resources of mobile devices. In some cases however, battery utilization might become secondary. This includes scenarios where detection accuracy is given a higher priority over battery utilization. Thus, a detection framework has to be intelligent and flexible. To reach this goal, our framework relies on building multiple scalable context based models, and observing the behaviour patterns of Android devices by comparing to relevant pre-built models. We make use of machine learning classifiers that are more scalable to

help classify features that could be used to detect malware by behaviour analysis. In this framework, the expensive analysis components utilizing machine learning algorithms are pushed to server side, while agents on the Android client are used mainly for context-aware feature gathering to transmit the information to server side classifiers for analysis, and to receive classification results from the server side agents.

Preface

This thesis presents research conducted by Mohammed Shahidul Alam under the supervision of Dr. Son Vuong. All of the research results available here have been published in peer reviewed conference proceedings or in a book chapter.

The research work presented in Chapter 3 was published in [2, 4]. The initial idea to use agent based systems for intrusion detection using machine learning approaches was published in [77].

The research work in [4] provided a description of the agent platform, and how the agents interact with each other using agent communication language in the Android mobile platform. The research work in [2] expanded on it to provide details on how the machine learning models stored on servers would be used by the agent based platform to analyze feature data and provide classification results.

[4] was joint work with Zhiyong Cheng, Dr. Vuong's MSc student. I was responsible for designing the initial framework to use JADE mobile agent platform for feature gathering on Android devices. Zhiyong Cheng was responsible for performing development of Agent Communication Language (ACL) and for initial testing on rooted Android devices. Details of his work has been published in [28].

The research work presented in Chapter 4 was published in [5] and [3]. The

research work in [5] provided initial experimentation details on a 5-fold cross validation approach on Android feature datasets. The research work in [3] expanded the previous work by adding 10 fold cross validation tests, and comparing the results to a separate training set and validation set. We also measured the space and time requirements to generate each model.

I have received the required permissions for using figures obtained from published sources.

The following are the list of publications related to this thesis:

- [4]: Mohammed Alam, Zhiyong Cheng, and Son Vuong. Context-aware multi-agent based framework for securing android. In *2014 International Conference on Multimedia Computing and Systems (ICMCS)*, pages 961—966. IEEE, 2014.

- [2]: Mohammed Alam and Son Thanh Vuong. An intelligent multi-agent based detection framework for classification of android malware. In *Active Media Technology (AMT)*, pages 226–237. Springer, 2014.

- [5]: Mohammed S Alam and Son T Vuong. Random forest classification for detecting android malware. In *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing*, pages 663—669. IEEE, 2013.

- [3]: Mohammed Alam and Son Thanh Vuong. Performance of malware classifier for android. In *6th Annual Information Technology, Electronics and Mobile Communication Conference*. IEEE, 2015.

- [77]: Son T. Vuong and Mohammed S. Alam. Advanced methods for botnet intrusion detection systems. In *Intrusion Detection Systems*. InTech, 2011.

Table of Contents

Abstract . . . . . **ii**

Preface . . . . . **iv**

Table of Contents . . . . . **vii**

List of Tables . . . . . **xi**

List of Figures . . . . . **xii**

Glossary . . . . . **xiv**

Acknowledgements . . . . . **xvii**

1 Introduction **1**

1.1 Thesis motivation 1

1.2 Research questions and contributions 3

1.3 Organization of the thesis 6

2 Background And Related Work **7**

2.1 Android security 7

2.2 Agent systems 9
2.2.1 Security using multi-agent systems 10
2.3 Machine learning 12
2.3.1 Machine learning for Android malware detection 13

3 Context-aware Multi-agent Framework For Securing Android . . . 17
3.1 Introduction 17
3.2 Related work 19
3.2.1 Context-aware mobile applications 19
3.3 Agent-based framework 20
3.3.1 System architecture 20
3.3.2 Agent categorization 24
3.3.3 Agent types 25
3.3.4 Agent communication language 33
3.4 Implementation 34
3.5 Evaluation 42
3.5.1 Battery context test 44
3.5.2 GPS context test 47
3.5.3 Network context test 50
3.5.4 Android API context test 51
3.5.5 Rooted device context test 52
3.5.6 Malware detection test 54
3.5.7 Verify Correlation agent can correlate 2 devices exhibiting similar malware behaviour 56
3.5.8 Server workload measurement 57

3.6 Discussion 60
3.7 Summary 63

4 Random Forest Classification For Detecting Android Malware . . . 65
4.1 Introduction 65
4.2 Background 68
4.2.1 Android feature collection 68
4.2.2 Random forest 70
4.2.3 Machine learning classifier for Android mobile systems . 71
4.3 Experiment 72
4.3.1 Dataset description 72
4.3.2 Classification experiment description 76
4.3.3 Hardware 77
4.3.4 Software 77
4.3.5 Sample code 77
4.3.6 Results 79
4.3.7 Experiment results with a 5-fold cross validation 86
4.4 Discussion 90
4.5 Summary 94

5 Conclusion And Future Work 95
5.1 Thesis summary 95
5.2 Future work 97
5.2.1 Application of machine learning 98
5.2.2 Use of multi-agent systems 101

Bibliography 102

List of Tables

Table 3.1 Android Features Observed [5] 34

Table 4.1 Android Features Observed [5] 75

Table 4.2 Amount of space required to store Random Forest model files as depth of trees is increased for a forest of 160 Trees 83

Table 4.3 Amount of space required to store Random Forest model files as number of features measured is increased for a forest of 160 Trees 85

Table 4.4 Experimental Results for varying number of trees and number of random features with Trees allowed to grow to maximum depth 87

Table 4.5 Misclassification Comparison of Different classifiers 91

List of Figures

Figure 3.1 JADE split-container runtime execution mode. 23

Figure 3.2 Interaction between client and server Agents. 33

Figure 3.3 CPU utilization comparison for battery levels. The x-axis displays the seconds that have passed since the application was started from left to right. The y-axis displays the CPU utilization in percentage of available CPU. [4, 28] 42

Figure 3.4 Network bandwidth utilization comparison for two battery levels. The x-axis displays the time that has passed since the application was started from left to right. The y-axis displays bandwidth utilization measured in KiloBytes per second. [4, 28] 43

Figure 3.5 Memory utilization comparison for two battery levels. The x-axis displays the seconds that have passed since the application was started from left to right. The y-axis displays memory utilization in Bytes. [28] 43

Figure 4.1 Classification rate of original dataset as the number of features are changed for forest of 10 trees for y range 80 to 96 percent 80
Figure 4.2 Classification rate of original dataset as the number of features are changed for forest of 160 trees for y range 80 to 96 percent 81
Figure 4.3 Classification rate of original dataset when tested on validation set for forests of 10 trees. The results show that as the depth of tree increases, the classification results are poorer due to over-fitting. 82
Figure 4.4 Classification rate of original dataset when tested on validation set for forests of 160 trees 83
Figure 4.5 Classification rate on model generated by SMOTE with testing on a validation set with forest of 160 trees. This graph shows the performance of the Random Forest algorithm on a balanced dataset. 84
Figure 4.6 Classification rate on model generated by SMOTE with 10 Fold cross validation testing with forest of 160 trees 85
Figure 4.7 Misclassification comparison with 20 trees as depth of tree is varied 88
Figure 4.8 Out Of Bag error rate comparison with 40 trees as depth of tree is varied 89
Figure 4.9 Root Mean Squared error rate comparison with 80 trees as depth of tree is varied 90

Glossary

ACL Agent Communication Language

ADB Android Debug Bridge

AID Agent Identification

AMS Agent Management Service

API Application Programming Interface

ARM Advanced RISC Machine

AVD Android Virtual Device

CFG Control Flow Graph

CPU Central Processing Unit

CONUCON Context-aware usage control

DCL Dynamic Code Loading

DDMS Dalvik Debug Monitor Server

DEX Dalvik Bytecode

DF Directory Facilitator

FIPA The Foundation for Intelligent Physical Agents

GPS Global Positioning System

GUI Graphical User Interface

IDS Intrusion Detection System

IPC Inter Process Communication

JADE Java Agent Development Framework

JICP Jade Inter Container Protocol

JSON JavaScript Object Notation

LEAP Lightweight Extensible Agent Platform

MMS Multimedia Messaging Service

NFC Near Field Communication

OOB Out-Of-Bag Used to measure the error rate in case of the Random Forest Classifier

P2P Peer To Peer

PDA Personal Digital Assitant

REST REpresentational State Transfer

RMA Remote Monitoring Agent

SMOTE Synthetic Minority Oversampling Technique

SMS Short Message Service

SVM Support Vector Machine

TCP Transmission Control Protocol

UI User Interface

UID User Identification Used in Android to partition access permissions of one application from another

XML Extensible Markup Language

Acknowledgements

First and foremost, I would like to thank my parents and my wife, for their unconditional love and support during the completion of this thesis. They were my support throughout my PhD studies during very difficult times, and helped me take major decisions, to keep me going. I will be forever in their debt. I would like to dedicate this thesis to them.

The amount of appreciation, love and respect I have for my supervisor, Dr. Son Vuong cannot be measured in words. He was my guide over the years and kept pushing me to try my best during the years that I have spent with him as his student. I would like to thank him for believing in me, and guiding me in my studies, exposing me to various industry-based research opportunities and on making life choices.

To both my children, thank you for being there, with your smiles, to keep me motivated during difficult times, and to remind me what life is all about.

To my committee members, thank you for pointing me in the right direction with my thesis, and for the valuable feedback that you provided.

I would like to thank all my colleagues in the Networks and Internet Computing laboratory (NICLab) for their insight and valuable feedback. My special

thanks to Jonatan Schroeder for being a close friend, providing valuable feedback regarding research and being there together during our struggles to complete our PhD programs.

I thank the almighty Allah for allowing me to complete this journey, and for teaching me more about life thorough my struggles.

Chapter 1

Introduction

1.1 Thesis motivation

Internet connected smartphones play an important role in our day to day communication needs. They are not only used for traditional cost incurring activities such as long distance phone calls and Short Message Service (SMS), but for a variety of other tasks that contribute to the domain of Internet-Of-Things. Today, smartphones are used to browse personal and corporate accounts on social networks such as Facebook and Twitter; conducting monetary activities such as paying for goods and services via credit card information saved on mobile devices using Near Field Communication (NFC) [33]; paying for parking using pay-by-phone applications [61]; and are used by security conscious users as 2-factor authentication devices for banking, email and cloud repositories such as Dropbox and Gmail. As such, malware authors have begun focussing their interest on coding malicious application for smartphones similar to what has been done for personal computing devices over the last three decades.

The most popular smartphone platform today is the Android platform developed by Google. Three out of four smartphone devices shipped today are based on the Android operating system. Over a billion devices based on the Google Android platform have been deployed since 2008. In Android, each application has an associated .apk file which is synonymous to a .exe file on the Windows platform. Due to the open software installation nature of Android, users are allowed to install any executable file from any application store. This could be from the official Google Play store [43], or a third party site. This ease of installing applications compared to other smartphone platforms such as Apple's iOS [15] platform makes Android users vulnerable to malicious applications. Moreover, unlike the iOS platform, initially Google did not verify if applications in their official Play store have malicious intent prior to making it publicly available. As such, malware had been found in the official Google Play store applications by security vendors [75]. The platform allows software downloads from third party sites; allows loading additional code at runtime Poeplau et al. [62] using Java-reflection or Dynamic Code Loading (DCL); and does not support patching vulnerabilities in older Android devices. As such, malware authors have predominantly focused on the Android platform. Today, it is the most vulnerable mobile platform with over ninety eight percent of malware built for it [27].

A recent vulnerability with the Android system includes the Stagefright library bug that leaves 950 million Android phones vulnerable to attack by a single Multimedia Messaging Service (MMS) that is undetectable by a user of the phone [1]. Devices based on the Android platform can be infected to send SMS to premium rate numbers to cause financial harm to users [76], used as part of botnets to cause

[1] http://blog.zimperium.com/experts-found-a-unicorn-in-the-heart-of-android/

distributed denial of service attacks [79], and steal banking credentials when these devices are used for 2-factor authentication [35]. Android devices can be pre-fitted with rootkits to track sensitive activities such as keystrokes and SMS messages [65]; and recently researchers have proposed the use of context-aware Android malware [47] that makes use of sensors on the device to trigger malware. As can be observed, many of the security threats identified with the PC industry have made their way into the mobile operating system space. A broad characterization of Android malware was presented by Zhou and Jiang [82] that provides the observed trends of malware for Android. They presented over 1260 malware samples in 49 malware families with a best-case detection rate of 79.6 percent and a low detection rate of 20.2 percent by commercial antivirus companies. Many of the malware samples diagnosed by them used drive-by-download attacks and update attacks similar to infection vectors used for personal computing devices.

1.2 Research questions and contributions

Most of the security solutions that have been proposed in literature requires modification of the Android operating system framework. In this thesis we propose the use of a multiagent system framework that attempts to detect Android malware from user space, without modification to the Android Operating System. The only addition is the optional addition of rooting of Android devices, which is an acceptable method in the Android environment. The primary base for making decisions however is using a machine learning approach. We make use of the Random Forest machine learning algorithm [22] to make decisions.

In this thesis, we try to answer the following research questions:

- How could existing machine learning algorithms be applied on features col-

lected for the detection framework for Android in novel ways?

- Would it be possible to detect Android malware without modifications to the Android framework? i.e. Could features be monitored by an Android application (at user level) to detect malware without modifying the Android Operating System?

- Could context awareness of Android devices, or feature correlation between Android devices be used to better detect Android malware? How could this be enabled given the resource constraint nature of mobile devices and the ever changing behaviour of mobile malware?

The following are our thesis research contributions:

- Contribution 1: We have identified that the machine learning classification algorithm Random Forest is a good candidate for detecting Android malware data features. This algorithm is a multi-class classifier that is robust to inter-dependence between the features that have been collected while monitoring a system, and the number of features that are observed to make a classification decision. This algorithm requires observing log m features to yield results, where m represents the total number of features observable in the system. This allows the algorithm to compute a decision faster. We published 2 papers that modify the hyper parameters of the algorithm on feature vectors of Android to classify Android malware. The results of our papers are discussed in Chapter 3. In the first paper [5] we provided initial experimental results by performing just a 5-fold cross validation. In our second related paper [3], we performed both 10-fold cross validation experiments and separate training - validation set comparisons by modifying the parameters of

the Random Forest algorithm. Moreover, we performed 3 experiments with random seeding for each setting of the algorithm and reported the median values.

- Contribution 2: We identified that machine learning algorithms are highly sensitive to parameter settings. There is a significant difference in the detection accuracy of a machine learning algorithm, such as Random Forest if it is used with the default parameters, as is done in most literary work, versus manipulating the parameters of the algorithm. In our experimentation, we varied the number of trees in a Random Forest algorithm, the number of random features compared at each decision point, and the depth of each tree. We find significant difference in results. We also find a variation in the detection rate while performing a 10-fold cross validation and validation set test. A 10-fold cross validation provided a 96.40 percent detection rate whereas a validation set provided 81.64 percent detection rate. Thus, we observe that the results obtained are very sensitive to the parameter settings.

- Contribution 3: We have used Java Agent Development Framework (JADE), an agent based middleware, to design our detection framework, to gather relevant features from Android devices. We have not come across any other research work that uses multiagent systems for detecting malware on Android devices. Our current system uses reasonable amount of Central Processing Unit (CPU) and memory on Android devices to gather context aware features. We emphasize the use of context-awareness as many mobile malware today are launched based on the network the user is connected to, or region that a user is located in physically, or the version of the operating system

the user uses. Our architecture pushes the model generation and detection component to server side as it is a more computationally intensive task. We provide work-load statistics on the server such as CPU time, and memory utilization in running the agent-platform.

1.3 Organization of the thesis

The rest of the thesis is organized as follows. In Chapter 2, we provide related background work to this thesis. In Chapter 3, we provide the design and experimentation results of using a multi-agent environment for collecting Android-based features. In Chapter 4, we propose the use of the Random Forest machine learning algorithm on an Android dataset. We perform extensive experiments and provide our results. Finally in Chapter 5, we summarize the thesis and include possible extensions to this research based on emerging research in the area of machine learning approaches to malware detection on Android.

Chapter 2

Background And Related Work

In this chapter, we review relevant background and related work that are foundational to this research work. This thesis addresses the subject of Android security using multi-agent systems as a detection framework. The multi-agent system uses a machine learning algorithm for detection and analysis of Android malware. Thus, we provide background information on the following three research domains:

1. Android security
2. Agent systems
3. Machine learning

2.1 Android security

Research and development for malware detection for Android mobile systems can be divided into two primary types. Static malware detection, and dynamic malware detection. Work in static detection is done by reverse engineering Android

executable files (.APK) using tools such as Androguard [32] to statically analyze the .dex byte code of Android executable files, or by signature matching by listening to executable installations on Android by monitoring Android intent messages for file downloads. Dynamic detection of malware uses monitoring of device behaviour either by monitoring the state of the system by inspecting the sensors on Android devices such as screen being turned off while an SMS is being sent, or by monitoring various system parameters. Given the extensive size of the data that is to be processed, many solutions now use machine learning approaches to solving the problem instead of manually working with available data.

Given that most malware threats have been targeted for Android systems, many approaches have been proposed in literature for securing Android devices. TaintDroid [37] modified the Android Dalvik virtual machine code to taint and track sensitive data stored on the device by causing a 32 percent overhead on CPU. They tracked the flow of user private information. Kynoid [68] extended the solution to provide real-time security policy enforcement for Android by using user-defined security policies defining temporal, spatial and destination constraints on data. Nauman et al. [60] provided an extended Android package installer allowing users to have more control over runtime constraints on Android applications by allowing fine grained access control policies on applications and adding constraints on resources allowed to be used. Shabtai et al. [70] classified Android threats into five categories and recommend incorporating Linux security solutions using SELinux [69]. Their evaluation results show significant performance degradation on CPU and memory usage. Limited resource issues were also faced by Schmidt et al. [67] when they recompiled various Linux tools to enhance Android security. As mentioned previously, most of these approaches require modifications to the base

Android framework which makes the solutions difficult for easy deployment.

2.2 Agent systems

The term agent or software agent is usually deciphered well in the artificial intelligence community, where it stands for a program that can behave autonomously to perform a multitude of dynamic tasks based on the logistics that have been programmed into it by a user. Advantages of using multi-agent platforms include [21]: asynchronous autonomous interactions between agents, easy software upgrades, and ability to function in heterogeneous environments.

Asynchronous autonomous interaction: This advantage is vital in a network where network connections are volatile, such as wireless networks. Even if the connection breaks, the agent could continue processing data on the mobile device and report back whenever the connection is reestablished. This adds to the agent's capability to work in a fault tolerant mode.

Software Upgrades: Usually in order to update software on multiple hosts, an administrator has to first stop the server functionality, then uninstall the old version of the software, and then reinstall the new version. The entire software system has to be stopped for upgrades. The advantage of agents in general in this situation is that if each component of the upgraded software is managed by an agent, then it is as easy as disabling the old agent and deploying a new agent which has the required functionality. In this way one could avoid bringing down the entire system and instead stop just a single agent-based component.

Functionality in heterogeneous environments: Most agents today can work in heterogeneous environments. This is due to the fact that these agents are usually written in a language which is portable to multiple platforms, such as java or perl.

Since agents sit on top of an agent framework, they can easily function regardless of if the host runs a version of Linux or Windows operating system. The significant reduction in costs of placing agent frameworks in hosts over the past few years have added to the benefits of running agents.

A disadvantage is the need for an agent platform to be supported by the underlying operating system (such as Android) for agents to deploy and communicate with each other. Communication between agents is achieved using The Foundation for Intelligent Physical Agents (FIPA) [1] compliant Agent Communication Language (ACL). This is the primary reason in choosing JADE agent platform as it is continually supported on Android devices. Other platforms supported for Android include JaCa-Android [66].

2.2.1 Security using multi-agent systems

The use of multi-agents for security had been done earlier for computers. We now discuss some related work in the field.

The earliest work in this area was started by Purdue University's CERIAS (The Center for Education and Research in Information Assurance and Security) group when they put forward a proposal for building an autonomous agent based security model by using genetic programming [29]. This was followed up by their work in implementing the earlier proposal [19]. This system was called AAFID (Autonomous Agents for Intrusion Detection) written earlier in Perl, Tcl/Tk and C, and later revised and written in the perl language to make it more portable. Helmer et al. [48] used an anomaly detection technique by using the Ripper algorithm on sendmail system calls. The architecture mimicked a portion of the Java Agents for

[1] www.fipa.org

Meta-Learning (JAM) project [73]. A distributed hierarchical Intrusion Detection System (IDS) was proposed by Mell and McLarnon [55] that tries to randomize the location of agents and decentralizing directory services. The system also resurrects agents killed by an intruder as there always exists multiple copies that track the original agent and vice versa. The Micael IDS was proposed by [31]. They proposed an additional feature of periodically checking if all agents are active in the system. Another prominent work that detects intrusions using mobile agents is the IDA system [16]. This system tries to backtrack intrusion attempts by looking into MLSI (Mark Left by Suspected Intruders) left at each host. They also emphasize tracking the steps that an attacker takes.

The Sparta system by Kruegel et al. [50, 51] is the most extensive work done till date on using mobile agents and intrusion detection. Sparta, which stands for Security Policy Adaptation Reinforced Through Agents, is an architecture that is capable of monitoring a network to detect intrusions and security policy violations by providing a query like functionality to reconstruct patterns of events across multiple hosts. This is a network-based IDS that correlates data from multiple sensors located throughout the network. The authors have created an EQL (Event Query Language) with syntax similar to SQL (Sequence Query Language) used in databases.

Other mobile agent based IDS's include a Peer To Peer (P2P) based IDS [63] that works in a neighbourhood watch manner where each agent looks after other agents in its vicinity by using a voting procedure to take action against a compromised agent; the MA-IDS system [52] which uses encrypted communication between the mobile agents in the system, and use a threshold mechanism to detect the probability for each intrusion depending on the quantity of each intrusion

type obtained allowing it to learn in a one dimensional method. Some other agent based IDS's include the work by Foukia et al. [39, 40] which uses a social insect metaphor and immune systems to model an intrusion detection system.

2.3 Machine learning

The domain of machine learning is used in solving problems where the problem size is large enough such that it would be difficult for humans to adequately make sense of data. Machine learning algorithms are used to extract features from the data such that it could be used for solving predictive tasks such as classifying, decision making and forecasting results [30]. Since this thesis uses *predictive* or *supervised learning* approach, we provide basic meaning of such a method. In supervised learning, given a labeled set of input-output pairs (the training set) with N training examples, the goal is to predict a future y value given an input. The input to such a system is x_i which is a D-dimensional vector, also called a feature vector. x_i is a set of values used to determine y_i. When y_i is categorical (such as malware, goodware, etc.), the problem is considered as classification [58].

Classification techniques such as Support Vector Machines, K-Nearest Neighbours, Decision Trees, Logistic Regression and Naive Bayes have widely been used in the area of intrusion detection research in the security community. They are predominantly used for behaviour based detection methods, also called anomaly detection methods. We now present some related work in the use of machine learning approaches used for Android malware classification.

2.3.1 Machine learning for Android malware detection

Allix et al. [6] observed over 50000 applications including malware from the genome project by Zhou and Jiang [82] which is a dataset of Android malware used for research purposes. The authors created a set of features based upon textual representation of basic blocks extracted from Control Flow Graph (CFG) of application byte codes. They observed 2.5 million features. Since these features are computed statically, they will however miss dynamically loaded features in Android, made possible by using the Android *dexClassLoader* library. The authors compared C4.5, ripper, Support Vector Machine (SVM) and Random Forest algorithm with their default settings and conduct experiments by adjusting the good ware to malware ratio and the number of features to use.

Yury et al. in [81] addressed the problem of dynamic code loading in Android by using a client-server model where a client runs the dynamically loaded modules from an Android device emulator, and feeds it to a server that conducts static analysis. They use the concept of method call graph to capture new method calls generated by dynamic loading and pass it to the static component to add it to the list of features observed.

Glodek and Harang [41] created 19137 features from 2 or 3 permission sets obtained by setting a threshold value of 0.05 where the threshold defines the feature being present in malicious vs benign sample. They used the random forest algorithm with 25 trees and maximum depth and had results of 92 percent true positive and 3 percent false positive.

Kim et al. [49] developed an automatic feature extraction tool implemented in JavaScript for static detection. The input to their system is a .apk Android exe-

cutable file. The authors use JavaScript code to extract the following as features: Number of permissions requested by the application (gathered from the manifest.xml file in Android .apk package file); API count of each method related to phone management invoked from the code; API count of each method related to phone control and privacy information called from the code. Based on these features collected, the authors use *J48 Decision Tree* classifier from the Weka library and perform 10 fold cross validation on 1003 Android applications. They received a true positive rate of 82.7 percent and false negative rate of 17.3 percent.

Burguera et al. [24] used a rooted Android device. The Linux based tool *strace* was used to capture system calls. The authors captured the number of each system call invoked by a given Android application to form the feature vector. Each feature vector was composed of 250 linux 2.6.23 system calls. Based on experimental results, the most relevant system calls for malware detection were *read(), open(), access(), chmod() and chown()*. The authors used a simple 2-means clustering algorithm to distinguish between benign applications and their corresponding malware version. The distance between clusters is just a Euclidean distance between the feature vectors. Their solution would cluster similarly named applications into two different clusters, one for the malware and the other for the benign sample. The solution works under the assumption that a benign sample is re-fitted with malicious code before being uploaded. Thus, a malware without a benign version cannot be detected. One weakness of the system is that their tool cannot detect malware that uses very few system calls, as was exhibited in Monkey Jump 2 malware.

Dini et al. in [34] used a rooted device to design a host based real time anomaly detector based on 1-Nearest Neighbour classifier. They monitor 13 features. At

user-level 2 features are monitored: if the phone is active/inactive; and if SMS is being sent when the phone is inactive. At the kernel level, they created a kernel module to monitor 11 system calls: *open, ioctl, brk, read, write, exit, close, sendto, sendmsg, recvfrom, recvmsg*. Their model monitors system wide behavior using 2 models, one which captures the features every 1 second interval, and another which captures every 1 minute interval. The system was tested on 10 genuine malware samples. The authors claim a malware detection rate of 93 percent in general with a false positive rate of 0.0001.

Shabtai et al. in [71] monitored 88 features on an unrooted device. They monitor the features on 2 real devices used by 2 different users. They tested their approach on 16 benign applications and 4 self generated malware. Their model monitors features at 2 second intervals. The authors compared the use of the following classifiers: k-Means, Logistic Regression, Histograms, Decision Tree, Bayesian Networks and Naive Bayes. The authors use a *filter* approach for feature selection by comparing Chi-Square, Fisher Score and Information Gain. They used top 10, 20, 50 features for computing the scores for each feature out of the 88 features available. Based on their evaluation, they have a detection rate of 80 percent. Their false positive rate is 0.12. They claim that Naive Bayes and Logistic Regression were superior for most configurations. Fisher score with top 10 features scored the best among their experimentation. The authors identified the following features as being of importance in distinguishing malware and benign applications: *Anonymous_pages, Garbage_Collections, Battery_Temp, Total_Entries, Active_Pages, and Running_Processes*.

Amos in [10] created an automated system to analyze malware samples. The tool allows for the automated analysis of benign and malicious Android applica-

tions by running adb scripts in Linux systems. Since we used data gathered by [10] for our experiments, section 4.3.1 provides more details about the dataset. The author compared results from using the following classifiers: Bayes Net, J48 decision tree, Logistic Regression, Multilayer Perceptron, Naive Bayes and Random Forest. We would like to point out that Random Forest was only tested with the default setting of 10 trees and 6 parameters.

Z. Aung and W. Zaw [17] performed a static analysis of 500 Android .apk files by inspecting the permissions requested by a given application. They used 160 permissions as their feature vector. They compared Random Forest, J48 Decision Tree, and Classification and Regression Tree (CART) algorithms. Unfortunately they do not discuss the parameter setting used for Random Forest, nor the number of benign and malicious applications used for experimentation.

Chapter 3

Context-aware Multi-agent Framework For Securing Android

3.1 Introduction

Many commercial and research prototypes have been developed to deal with securing Android. Independent testing of commercial solutions show that less than half of them are efficient in detecting malware [54]. Many of the security solutions proposed in literature require modification to the Android framework. Moreover, the frequent release cycle of Android causes modifications to the underlying Android API, making many of the proposed security solutions no longer viable. Lack of available resources such as battery, memory and CPU on these devices further lead to constraints that need to be satisfied before traditional Linux security tools

can be directly applied.

Given the availability of multiple sensors on Android mobile devices that can be accessed by Android applications using Android API; and the lack of available on-device resources such as battery, memory and CPU; we propose the use of a context-aware multi-agent based framework to monitor and dynamically launch investigative agents on Android mobile devices. We provide a framework design, implementation details and evaluation of our proposed framework to show that a multi-agent based system is a viable option for securing such devices. We propose the use of spatial context-information gathered using location information(GPS, network) and device context (battery level, Android API level, rooted device) to make context-aware decisions for the framework. We also show that we can use context information to reduce frequency of network connections and CPU utilization overhead caused by the Multi-Agent framework.

According to our knowledge, we have not come across any other research work that uses multi-agent based systems on Android devices that make context-aware decisions to detect malware threats. Our primary goal is to gather context aware data from Android devices to choose which machine learning classifier model to use [5]. The main contributions of this chapter can be summarized as follows:

- Provide the design and implementation of a context-aware agent-based framework based on the Java Agent Development Framework (JADE) [74] multi-agent platform.

- Use both spatial context information gathered using location information (GPS, network); and device context (battery-level, Android API, rooted device detection) to make context-aware decisions by dynamically modifying

detection behaviour of deployed agents.

- Provide evaluation of the agent system applied to the Android context to evaluate system overhead incurred by the use of the detection framework.

- Provide experimental usage scenarios for detecting Android based malware by collecting features.

The rest of the sections are organized as follows. Section 3.2 provides background information needed to understand the problem domain. Section 3.3 provides the proposed multi-agent based framework; Section 3.4 provides implementation details; Section 3.5 provides initial results of experimenting with the system; Section 3.6 discusses concerns that might be raised with the proposed framework.

3.2 Related work

3.2.1 Context-aware mobile applications

Initial development based on JADE-LEAP was provided by Moreno et al. [57] for development of a taxi-cab service application on Personal Digital Assitant (PDA) using bluetooth communication. The closest related work to context-awareness in the Android operating system space for security was proposed by Bai et al. [18]. The authors proposed a context-aware methodology for securing Android operating system based on Context-aware usage control (CONUCON) model. Their system requires modification to the Android framework to hook the *checkComponent-Permission* function to monitor objects and associated permissions. Similarly, the MOSES system proposed in [80] also requires modifications to be made to the underlying Android operating system. The purpose of MOSES is to use context to

maintain security profiles and allow switching between profiles during runtime. In our system, we do not require modifications to be made to the Android framework. We optionally require devices to be rooted if privileged tools are to be used for packet inspections. Rooting a device is a more acceptable solution based on the popularity of rooted application in the Google Play store.

3.3 Agent-based framework

3.3.1 System architecture

The system that we have developed has a client-server model. A client-server model is necessary because of the pre-condition set by the Java Agent Development Framework (JADE) agent platform which requires the server to host two JADE specific services: the Agent Management Service (AMS), and the Directory Facilitator (DF). These services allow agents in the system to discover each other.

JADE platform services

AMS: Agents in the system have to register with the AMS. The AMS supervises the entire platform of agents. It checks the validity of the unique Agent Identification (AID) of an agent, and keeps track of the life-cycle of agents in the system. There exists a single AMS for the entire system. The AMS resides in the main container of the platform and communicates messages in the system using the *jade.domain.JADEAgentManagement.JADEManagementOntology* ontology class. Some of the management functionality performed by the AMS include:

1. *create-agent*: This action allows the AMS to create an agent of a given class type with a chosen agent name in a specific container.

2. *kill-agent*: This action allows the AMS to destroy the uniquely identified agent.

3. *kill-container*: This action allows the AMS to destroy an agent container.

4. *query-platform-locations*: This action allows the retrieval of all containers that are available in the platform.

5. *query-agents-on-location*: This action allows the retrieval of all agents in a given container.

6. *where-is-agent*: This action allows locating the containerID where the agent resides.

As can be seen, using a combination of queries, the location of all agents and their associated containers can be queried from the AMS. The AMS also allows any agent in the platform to subscribe to platform events using a FIPA-Subscribe protocol. Though we do not use subscription in our framework, some of the tasks that are allowed include: subscribing to agent state such as creation, suspension, resumption, movement, cloning and destruction; container creation and destruction; and platform shutdown requests.

DF: The DF is an optional yellow page service that allows agents in the system to advertise their services in the system and also query to find agents that provide a service. There exists zero or more DF services that can interact with each other as a federation. Some of the services allowed by the DF include:

1. *Registration*: Agents that want to publicize their capabilities have to locate a DF to register its agent description.

2. *Deregistration*: Agents can also request to de-register from a DF when it no longer wants to publicize its services.

3. *Description modification*: Agents can also modify their agent-description if they want to modify their advertised services.

4. *Search*: Agents can query the DF to look-up agents providing an advertised service through their registered agent descriptions.

Split container runtime execution mode

In JADE, every agent that runs in the system has to exist in a JADE runtime container. For mobile devices however, JADE also allows a split execution mode using the Lightweight Extensible Agent Platform (LEAP) add-on. The split-container model is recommended for resource constraint mobile devices. This approach runs a thin front-end client on the mobile device, and runs the back-end on a server. The front-end and back-end communicate using a dedicated connection that is robust to connection failures [20].

The communication setup between the front-end and the back-end is made possible by the use of a dedicated server in JADE called the *mediator*. The *mediator* runs at a well-known address that is network accessible by all mobile devices and by the various containers in the platform. During startup, the front-end sends a *CREATE_MEDIATOR* request using Jade Inter Container Protocol (JICP). The mediator then creates a back-end and connects the front-end to it. The back-end then tries to establish a connection with the main container hosting the AMS. Once the newly created split-container is registered with the main-container, the mediator notifies the front-end that the registration was successful. The mediator takes no

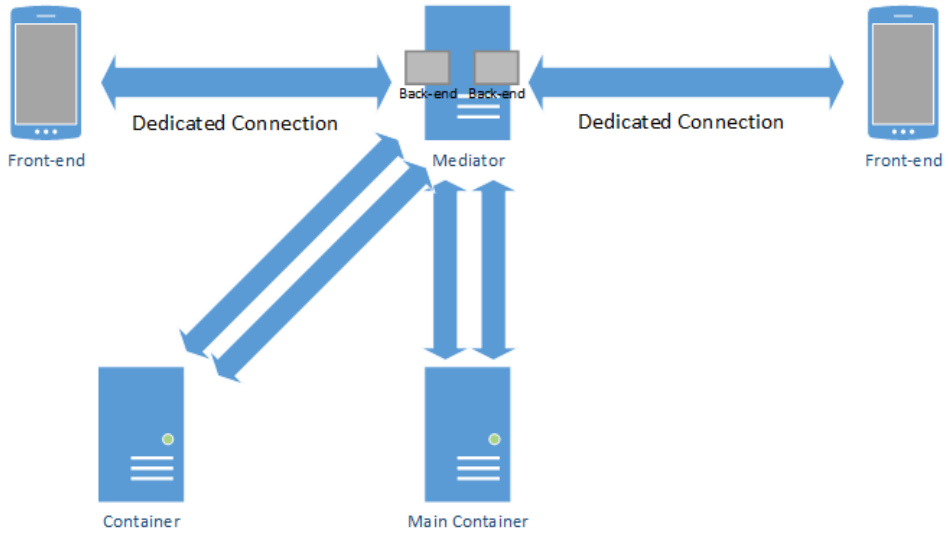

Figure 3.1: JADE split-container runtime execution mode.

part once the connection has been established between the front-end and the back-end. Fig. 3.1 shows the interaction between the various components just described.

Some of the advantages of the split-container mode include:

1. *Connection loss transparent to applications*: The front-end and back-end uses a store and forward methodology to transmit messages i.e. if there is a connection disruption between the front-end on the mobile device, and the back-end on the server, then all messages are buffered in both ends. When the connection resumes, the messages are exchanged.

2. *IP address of mobile-devices can be dynamic*: Since the agents on the mobile device interact with other agents in the platform using the back-end, a change in the IP address of the mobile device hosting the front-end causes no issues. Agents on other containers need to know the IP address of the back-end server only.

3. *Lightweight communication*: The front-end hosted on the mobile device is more lightweight than a full container as most of the communication with other containers is taken care off by the back-end.

Though there are advantages to using the split-container on mobile devices as mentioned, cloning of agents and mobility of agents are not supported in the split-container method. If such functionality is required, the JADE platform allows dynamic behaviours to be loaded into mobile devices running in a split-container execution mode by using the *jade.core.behaviours.LoaderBehavour* class.

In our framework, the Android devices are considered as clients. Each Android device hosts the front-end. Each Android device runs an Android application that is wrapped using a Jade split-container runtime service as described. We also use servers that host multiple framework specific service agents. The service agents perform more computationally expensive operations, and as such have not been placed on mobile devices to conserve resources. The agents launched on the server are detailed in section 3.3.3.

3.3.2 Agent categorization

Our agent-based framework consists of two categories of agents. The first type of agents can be categorized as the *data* or *feature collector* agents. These agents reside on Android devices to perform data collection tasks at various intervals of time. Some of these agents persistently reside on the device, whereas others are launched based on contextual information of the device. Some of the contextual information used includes: location of the mobile device; Android-API level of the operating system; device root status; and applications available on the device. The second type of agents are the *service* or *analysis* agents. These agents usually

consume higher resources, and are preferably placed on the server side. These agents are active for longer durations of time and are responsible for maintaining long term profile information of Android devices.

3.3.3 Agent types

Based on the design principles of JADE [56] agent-oriented development methodology, all users of the framework; all devices that exist in the framework; and all resources available in the framework are to be identified as agents. Each agent in turn, can then be further partitioned based upon their role as providing support service to other agents; help in agent and resource discovery; work as agent-framework management; or perform device or framework monitoring. As shown in Fig. 3.2, after these refinements, we have the following agent types with associated functionality.

Following are the agents that reside on the client side (Android device):

- **Profile agent:** This agent is active for the entire life cycle of the agent platform on the device. This is the primary agent on the client side. Some of the tasks performed by this agent includes:

    1. Responsible for communicating with the Profile Service provider on the server to receive commands.
    2. Maintains a local copy of the profile information of the Android device it is installed on.
    3. Register with the Profile Service provider on startup.
    4. Launches Action agents after communicating with the Location and Sensor agents on the Android device to collect context-aware data when requested.

5. Maintaining threshold levels to optimize use of device resources. Monitoring battery is one of those resources.
6. Performs notification tasks to inform the Profile Service agent about completed tasks. Providing computed feature vectors through pre-defined Agent Communication Language (ACL) ontology is one of the tasks.

- **Sensor agent:** The primary task of this agent type is to monitor sensors available on the Android device. The tasks performed by this agent includes:
  1. Subscribes to sensor event that occur on the Android device by using *intent* messages. This is required so that the agent is notified by the Android operating system of any changes that occur with network or sensor events on the device.
  2. Monitor static information of the Android device. This includes checking: if the device is rooted; and the Android version of the operating system that has been installed on the device.
  3. Monitor network information of the Android device. This includes checking the network type the device is connected to i.e. if it is connected to a wireless access point; the name of the wireless access point; the IP address assigned to the Android device.
  4. Reports any changes to the information gathered to the Profile agent on the Android device so that the Profile agent always contains updated information. This is required as the Profile agent will perform tasks based on context information received from the Sensor agent.
- **Location agent:** The primary task of this agent is to handle all location

related procedures. The tasks performed by this agent includes:

1. Subscribes to Global Positioning System (GPS) events using the *android.location.LocationListener* Android class. The LocationListener class is used to receive notifications sent by the *LocationManager* during reported location changes. The Android LocationManager reports to the Location agent as it registered with it during startup.
2. Directly communicates the location information to the Location Service agent on the server in terms of its latitude and longitude.
3. Directly communicates the location information to the Profile agent on the Android device.
4. Can retrieve list of nearby Android devices from the Location Service agent.

- **Action agent:** This agent type is the most important data collector agent in the framework. The tasks performed by this agent includes:

1. Dynamically launched by the Profile agent. The dynamic launch behaviour is caused either because one of the rules maintained by the Profile agent is triggered; or the Profile Service has asked the Profile agent for data, that can only be collected by launching an Action agent.
2. Maintains context-aware data. This is required as not all Action agents can perform its tasks without certain context information. Every Action agent maintains the Android API-level required for functionality. This is required as the functionality of the Agent might not be compatible with certain Android API-levels. For example, Android 4.3 allows

applications to access Android notifications by creating a notifications listener. This feature was not available in earlier API levels.

3. Stores its resource utilization level in terms of memory, CPU, disk and network utilization. A higher value denotes a higher resource utilization. Currently we have set it manually, but we foresee agents being able to automatically compute these values based on the specific device resource such as the amount of memory in the device and the processor used.

4. A particular Action agent runs on the Android device, if and only if the resource constraints and API constraints pass. The decision is made by interacting with the Profile agent on the device as it keeps track of all running agents on the device. The Action agent reports its resource utilization score to the Profile agent. The Profile agent verifies that the summation of resource utilization of all Action agents is below the threshold. If it is, then the Action agent is notified to perform its tasks.

5. Tasks performed by the Action agents include

   - monitor application installations
   - report memory and CPU usage
   - report binder API statistical values
   - IP addresses being communicated with and which services are causing the network traffic
   - detect if device is rooted
   - use tcpdump for Advanced RISC Machine (ARM) to capture network packets if the device is rooted and the tcpdump service exists

on the mobile device

Now that we have described all the agents on the client side, we now provide details of the tasks performed by the agents on the server side. The server-side agents are also called service agents. It should be clarified that not all the server side agents have to reside on the same physical machine. Given the existence of AMS and DF agents in the platform, they can be queried to locate the relevant server side agents. The agents communicate with each other using pre-specified Agent Communication Language (ACL) messages.The server side agents in our framework are:

- **Subscription Service agent:** The primary task of this agent is to maintain historical information of connected Android devices and their state. The tasks performed by this agent includes:

  1. Register Android devices when they join the framework.
  2. De-register Android devices when they want to exit the platform. Note that since the Android devices in our system use a split-container execution mode as discussed prior, the Android device would have to quit the Android application to disconnect the back-end from the framework.
  3. Maintains list of currently connected Android devices in the system.
  4. Maintains historical information about each connected Android device regardless of if it is currently connected or not.

- **Location Service agent:** The primary task of this agent is to maintain historical information of the location history of each Android device that has

ever connected to the system. The tasks performed by this agent includes:

1. Responsible for receiving location updates from Android devices through the Location agent on the Android device.
2. Maintains a time-stamped list of all reported locations for each Android device. This is useful for obtaining historical information of location for each Android device.
3. Responsible for providing list of nearby devices when queried by any agents in the system.

- **Profile Service agent:** The primary task of this agent is to maintain an event feed for all active Android devices. The tasks performed by this agent includes:

  1. Maintains the profile of every Android device. This includes both static and dynamic information of the Android device such as the API level of the Android device, if the device is rooted, the IP address assigned to the Android device, e.t.c.
  2. The Profile Service agent periodically receives updated profile information from the Profile agent on the Android device.
  3. This agent sends recommendations to Android device Profile agent to start new (Action) agents.
  4. This agent launches a Device Analysis agent for each Android device in the system, if one does not exist.
  5. This agent requests Profile agent on the Android device to modify monitoring behaviour based on feedback from the Device Analysis agent

associated with the Android device.

6. This agent relays the feature vectors sent by the Profile agent on the Android device to the Device Analysis agent for analysis / classification.
7. Maintains historical data regarding the number of malware feature vectors observed. This value is used to modify observation behaviours on the Android device by informing the Profile agent on the device.

- **Device Analysis agent:** The primary task of this agent is to verify if an Android device exhibits malware behaviour. There exists one Device Analysis agent for each Android device in the system. The tasks performed by this agent includes:

  1. This agent receives the feature vector to compare, and how it was computed from the Profile Service agent. The Device Analysis agent needs to know how the feature was computed, so that it can pick a subset of pre-generated models to compare against.
  2. This agent uses the machine learning classification models built using Weka to compare the feature vectors [5].
  3. Notifies the Profile Service agent if malware behaviour was observed from the sent feature vector. Our current design compares the feature vector against each relevant model stored in our system. If malware behaviour is reported by more than a set threshold number of models, only then is it reported to the Profile Service agent.
  4. Maintains historical information of feature vectors observed to be malware, and models that marked it as malware. This information is useful

to correlate devices that exhibit similar maliciousness by the Correlation agent.

- **Correlation agent:** The primary task of this agent is to correlate Android devices that exhibit similar malware behaviour. This agent is launched by a system administrator. The tasks performed by this agent includes:

  1. Locate all Device Analysis agents in the framework. This is required as the Device Analysis agents contain data related to:
     - If the associated Android device has exhibited malware behaviour.
     - The feature vector that caused the malware behaviour.
     - The machine learning models that caused the detection.
  2. Query the Device Analysis agents if it detected malware on a particular machine learning model. If yes, the Device Analysis agent sends the unique name of the Android device and the feature vector that caused failure on the model.
  3. Query the Device Analysis agents if it detected malware on a certain number of models. If yes, the Device Analysis agent sends the unique name of the Android device and a key:value pair of the model and feature vector that caused failure on the model.
  4. Correlate all Android devices that have exhibited malware infection on the same model.

  Though the Correlation Agent performs basic operations now, greater logic can be included. This includes correlating Android devices infected in a similar location using information obtained from Location Service agent;

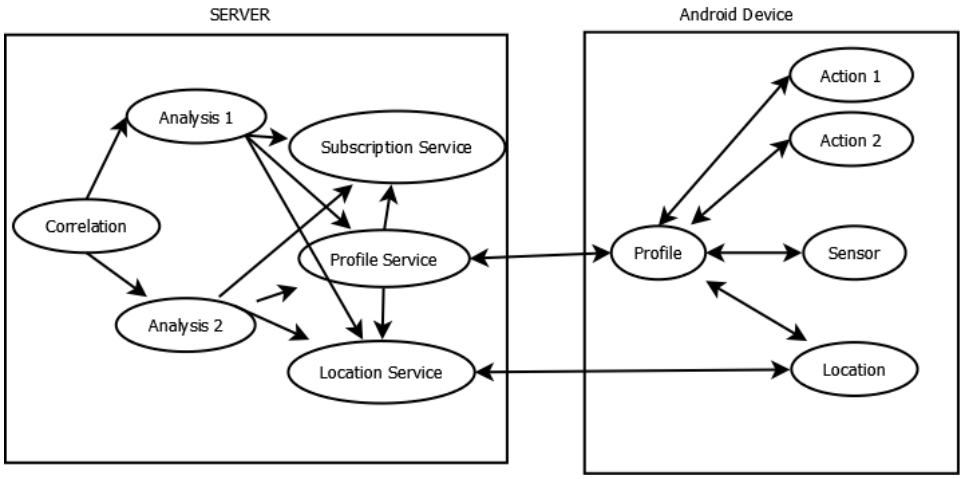

Figure 3.2: Interaction between client and server Agents.

or correlating Android devices infected in a particular time frame by using subscription information from the Subscription Service agent.

3.3.4 Agent communication language

Given that our data collection agents reside on Android devices with a split-container, and the service agents are placed in a server on a Linux or Windows installation, the mechanism for communication between the agents is via a FIPA-compliant Agent Communication Language (ACL). We developed an application specific ontology using the SL codec of JADE to compose ACL message concepts, predicates and agent actions [28]. *Concepts* are entity classes that encapsulate the values that need to be stored such as the resource threshold of the device, geo-coordinates of the device. *Predicates* are propositions which have to be verified such as registering and de-registering an Android device. *Agent actions* indicate tasks performed by agents in the system such as updating the location of android devices to the server, and querying nearby devices.

Table 3.1: Android Features Observed [5]

Category	Feature
Battery	IsCharging, Voltage, Temperature, BatteryLevel, BatteryLevelDiff
Binder	Transaction, Reply, Acquire, Release, ActiveNodes, TotalNodes, ActiveRef,TotalRef, ActiveDeath, TotalDeath, ActiveTransaction, TotalTransaction,ActiveTransactionComplete, TotalTransactionComplete, TotalNodesDiff, TotalRefDiff, TotalDeathDiff, TotalTransactionDiff, TotalTransactionCompleteDiff
CPU	UserCPU, SystemCPU, IdleCPU, OtherCPU
Memory	memActive,memInactive, memMapped, memFreePages, memAnonPages, memFilePages, memDirtyPages, memWritebackPages
Network	TotalTXPackets,TotalTXBytes, TotalRXPackets TotalRXBytes, TXPacketsDiff, TXBytesDiff, RXPacketsDiff, RXBytesDiff
Permission	Permissions

3.4 Implementation

In order to build and test our prototype system, we used JADE version 4.2. This was the most current version of JADE at time of implementation. We built an Android application that was set for Android API version 4.0.3. In the Android application, we used the jade-android library available for JADE version 4.2. We followed the procedures provided by authors of the JADE system in [25]. As mentioned previously, we used a JADE split-container execution mode in our framework. This is done by using the *jade.android.MicroRuntimeService* service class provided in the jade-android library provided. To use the MicroRuntimeService, the Manifest.xml file in Android needs to be modified to read:

```
<application ...>
```

```
    <service android:name="jade.android.MicroRuntimeService" />

</application>
```

This allows the JADE split-container runtime to be wrapped by an Android service through Android code.

The application specific ontology for Agent Communication Language (ACL) communication between agents was developed using *Protege* [59] and *Ontology-BeanGenerator* [1]. The OntologyBeanGenerator allows to create JADE ontology using Protege to create the necessary classes. There exists only one instance of the created ontology as a singleton pattern is used. The following code snippet shows how our ontology JASOntology is declared in an agent:

```
private Codec slCodec = new SLCodec();
private Ontology jasOntology = JASOntology.getInstance();

protected void setup()
{
            ...
ContentManager contentManager = getContentManager();
contentManager.registerLanguage(slCodec);
contentManager.registerOntology(jasOntology);
contentManager.setValidationMode(false);
          ...
}
```

More details regarding ontology generation and ACL message creation are provided in our related work [28].

In order to test our JADE-based framework, we used both a rooted Android Virtual Device (AVD) emulator and a physical rooted Android device as clients. The server side Service agents mentioned previously are located in a network reachable machine on the network.

To test on a rooted Android Virtual Device (AVD) for testing the use of deep packet inspection applications such as tcpDump for ARM [13], we installed *superuser* [12]. Superuser is a software package that allows Android applications to have access to root shell. Without this program, agents on the Android emulator cannot access the root shell.

We used *Busybox* [11] for using *uname, netstat* commands. The Busybox executable contains commonly used Linux utilities that is not available in Android by default. The tool has been optimized for use in embedded Linux environments.

In order to test if an Android device is rooted, we used the *RootTools* [38] application.

To find context information such as the API level of a device, we use the value from *android.os.Build.VERSION.SDK_INT*. We retrieve GPS locations by receiving updates from Android *LocationManager* using the *requestLocationUpdates* function. In order to access network related information, our application requests coarse, fine, and internet network permissions in the Android manifest file. This is done using the following code:

```
<uses-permission ..."...INTERNET" />
<uses-permission ..."...ACCESS_NETWORK_STATE" />
```

```
<uses-permission ..."...ACCESS_FINE_LOCATION" />
<uses-permission ..."....ACCESS_COARSE_LOCATION" />
```

For retrieving values mentioned in Table 3.1 used by our Action agent we read system files on Android. Following is a description of the features obtained:

The battery related feature information was collected by using an *intent* object on the BatteryManager class. The battery-based features measured are:

- *IsCharging*: Use the BatteryManager class of Android to check if the device is currently connected to a power source and in a charging state
- *Voltage*: Use the BatteryManager class of Android to check the current Voltage of the battery
- *Temperature*: Use the BatteryManager class of Android to check the current temperature of the battery
- *BatteryLevel*: Use the BatteryManager class of Android to check the current amount of battery power remaining (in range 0—100)
- *BatteryLevelDiff*: Change in battery level since the last time the feature vector was computed

All processes in Android communicate with each other using Inter Process Communication handled by Binder. The binder related feature information was collected by reading /sys/kernel/debug/binder/stats or /proc/binder/stats depending on the kernel version. The binder-based features measured are:

- *Transaction*: Number of Inter Process Communication (IPC) transactions performed

- *Reply*: Number of Reply Transaction messages processed
- *Acquire*: Number of Acquire Transaction messages processed
- *Release*: Number of Release Transaction messages processed
- *ActiveNodes*: Nodes (Processes) that are active
- *TotalNodes*: Total Nodes (Processes) maintained by Binder
- *ActiveRef*: Active references to binder objects
- *TotalRef*: Total References to binder objects
- *ActiveDeath*: Total Active processes killed and resources cleared
- *TotalDeath*: Total Processes killed and resources cleared
- *ActiveTransaction*: Number of Active IPC Transactions open
- *TotalTransaction*: Total number of Transactions performed since startup
- *ActiveTransactionComplete*: Number of Active Transactions that have been completed
- *TotalTransactionComplete*: Total number of transactions that have been completed
- *TotalNodesDiff*: Difference between current reading of TotalNodes and the previous reading of TotalNodes
- *TotalRefDiff*: Difference between current reading of TotalRef and the previous reading of TotalRef

- *TotalDeathDiff*: Difference between current reading of TotalDeath and the previous reading of TotalDeath
- *TotalTransactionDiff*: Difference between current reading of TotalTransaction and the previous reading of TotalTransaction
- *TotalTransactionCompleteDiff*: Difference between current reading of TotalTransactionComplete and the previous reading of TotalTransactionComplete

The CPU related feature information was collected by running the Linux *top* command every 1 second. The CPU-based features measured are:

- *UserCPU*: CPU usage by User processes
- *SystemCPU*: CPU usage by System processes
- *IdleCPU*: Idle percentage of CPU usage
- *OtherCPU*: Other CPU usage

The memory related feature information was collected by observing the *proc* directory in the underlying Linux system of Android. The files that were read are /proc/meminfo and /proc/vmstat. The memory-based features measured are:

- *memActive*: The total amount of memory being used as per /proc/meminfo
- *memInactive*: The total amount of page cache memory that are available as per /proc/meminfo
- *memMapped*: The total amount of memory used to map files, libraries or devices as per /proc/meminfo

- *memFreePages*: Virtual memory free pages as per /proc/vmstat
- *memAnonPages*: Virtual memory anonymous page count as per /proc/vmstat
- *memFilePages*: Virtual memory file system page count as per /proc/vmstat
- *memDirtyPages*: Virtual memory dirty page count as per /proc/vmstat
- *memWritebackPages*: Virtual memory write-back page count as per /proc/vmstat

The network related feature information was collected using the TrafficStats Android package. The network-based features measured are:

- *TotalTXPackets*: Total number of transmitted packets obtained from android.net.TrafficStats
- *TotalTXBytes*: Total number of transmitted bytes obtained from android.net.TrafficStats
- *TotalRXPackets*: Total number of received packets obtained from android.net.TrafficStats
- *TotalRXBytes*: Total number of received bytes obtained from android.net.TrafficStats
- *TXPacketsDiff*: Difference in value between current TotalTXPackets, and the previous reading
- *TXBytesDiff*: Difference in value between current TotalTXBytes, and the previous reading
- *RXPacketsDiff*: Difference in value between current TotalRXPackets, and the previous reading

- *RXBytesDiff*: Difference in value between current TotalRXBytes, and the previous reading

Android maintains the list of active applications in the system and the associated permissions requested by the applications. The permission-based feature measured is:

- *Permissions*: This feature is the summation of the total permissions requested by all running applications on the mobile device

In order to test the location-based context awareness, we programmed a helper Android application to generate time stamped mock GPS locations that is compatible with Dalvik Debug Monitor Server (DDMS) [45]. This application generates location events every 1 to 5 seconds using a time-stamped location dataset. The location update information in generated at random intervals to mock that users change their location at random. This allowed us to test launching of action agents based on the locations reported by the Location agent on the Android device. Dalvik Debug Monitor Server (DDMS) is a debugging tool which allows to monitor the system resource utilization of Android applications. Some of the relevant tasks that the tool allows includes:

- Screen capture of the device
- Radio state information of the device
- Location data spoofing
- Thread and heap information of the device
- Causing Garbage Collection

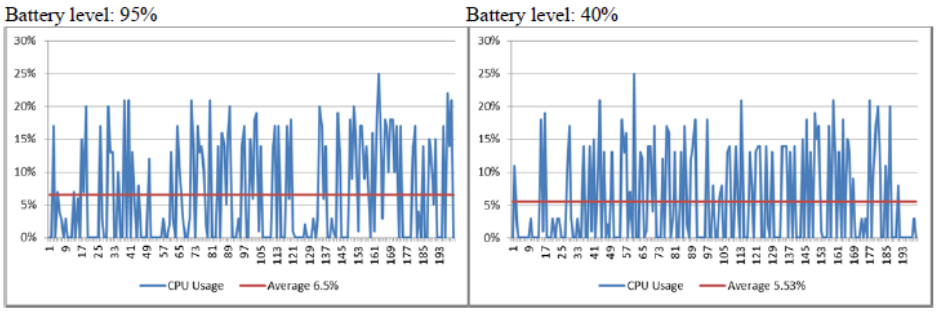

Figure 3.3: CPU utilization comparison for battery levels. The x-axis displays the seconds that have passed since the application was started from left to right. The y-axis displays the CPU utilization in percentage of available CPU. [4, 28]

- Tracking memory allocation of objects on the device
- Profiling individual methods of the application
- Using network traffic tool to monitor network usage

3.5 Evaluation

In order to evaluate the proposed framework, we run multiple experiments to capture the behaviour of the system based upon various contexts. The context-based tests that we describe next are based on:

1. Measuring the battery resource available (Battery context test)
2. Location of the mobile-device (GPS context test)
3. Network the device is connected to (Network context test)
4. Android API version of the operating system (Android API context test)
5. Device root status (Rooted device context test)

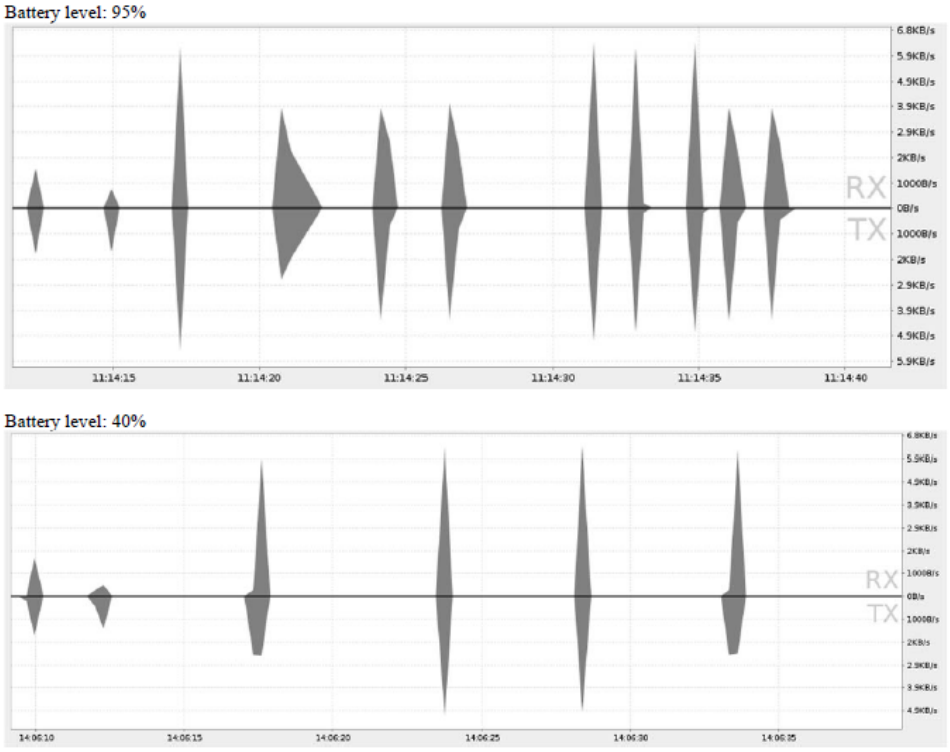

Figure 3.4: Network bandwidth utilization comparison for two battery levels. The x-axis displays the time that has passed since the application was started from left to right. The y-axis displays bandwidth utilization measured in KiloBytes per second. [4, 28]

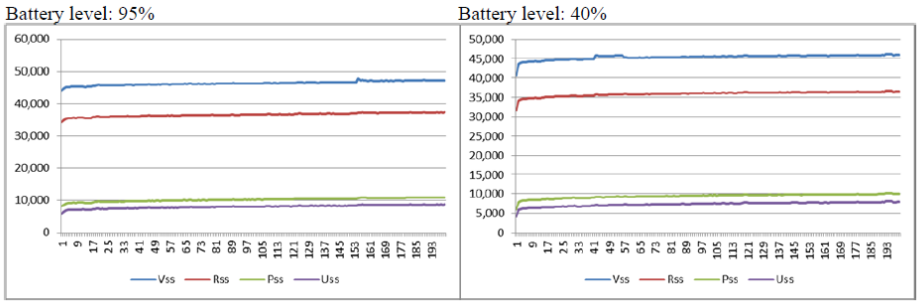

Figure 3.5: Memory utilization comparison for two battery levels. The x-axis displays the seconds that have passed since the application was started from left to right. The y-axis displays memory utilization in Bytes. [28]

It is to be noted that other context-based decisions could also be tested. Some of them include:

1. Application Context: What applications are installed on the mobile device, have they been marked as suspicious and are they in a running state;

2. IP based Context: What IP addresses are being communicated with and if any of them have been marked as suspicious;

3.5.1 Battery context test

For the Battery Context Test, we used a generic Android tablet containing an Allwinner A10 processor, Android 4.0.4 operating system installed, 512 Mb of physical memory and 4 gigabytes of storage space.

In order to test the changing behaviour of the system based on battery context, we conduct performance test runs to monitor the changes in CPU utilization, network usage patterns and memory usage for two different battery level contexts i.e. we use battery context to evaluate the system. We evaluate the system at a 95 percent battery level, and at 40 percent. In this test our system modifies its behaviour if battery is higher or lower than 50 percent.

The performance test run consists of the following steps:

1. Launch JADE agent platform on a network reachable server. This launches the JADE platform specific Directory Facilitator (DF) Agent and Agent Management System (AMS) agent; and the framework specific service agents: *Profile Service, Subscription Service, Location Service* in a main-container.

2. An Android agent application is next launched on the Android tablet mentioned previously. We provide the address of the network reachable server

activated in the previous step to allow the Android device to register itself with the Agent platform. Currently the mediator is hosted on the same machine where the main-container is placed. As previously mentioned, the mediator hosts the back-end of the split-container execution mode, whereas the front-end is hosted on the Android device.

3. We execute multiple scripts on the Android device using an Android Debug Bridge (ADB) terminal connected to the Android device to record CPU and Memory usage patterns on the Android device. The CPU utilization is measured using the Android Debug Bridge (ADB) command *adb shell top -d 1 -n 1* to measure CPU patterns at 1 second intervals. Similarly we use the *procrank* [36] tool to monitor a process's memory using the command *adb shell procrank* to monitor virtual memory set size (Vss), the resident memory set size (Rss), the proportional memory set size (Pss) and unique memory set size (Uss).

4. We monitor the network traffic by starting the network statistics tool in Dalvik Debug Monitor Service (DDMS).

5. We then connect to the JADE service on the Android device and create the relevant data collection agents: *Profile Agent, Sensor Agent, Location Agent*.

6. We collect the data results for analysis.

We run these steps twice. One when the battery was set at 95 percent level, and then again at 40 percent level. In order to change the reported battery level of the device, we used the *dumpsys* program through the Android Debug Bridge. It is to be noted that though we have used an actual device for the battery context test, the

same test could be run on an emulated Android device and modifying the reported battery level.

Some of the relevant dumpsys commands used include:

1. The following command provides battery information of the connected device: *adb shell dumpsys battery* The information listed includes:

```
Current Battery Service state:
AC powered: false
USB powered: true
Wireless powered: false
status: 2
health: 2
present: true
level: 100
scale: 100
voltage: 4100
temperature: 2200
technology: Li-ion
```

   Here, *level* lists the amount of battery available in the connected device.

2. The remaining battery level of a connected device can be set using the command: *adb shell dumpsys battery set level 40* Using this command, the battery level of the connected device has been set to 40.

3. To reset the battery level of the device connect we use the following command: *adb shell dumpsys battery reset* This command is required since once

a *set* command is run, the device no longer reports battery information from the actual device.

Using Action agents are not accounted for in the values as the behaviour of the system has wide variations in system performance based upon the tasks of the Action agent. For example, an Action agent that computes feature vectors every 2 seconds from multiple system files will have higher memory and CPU usage than one that computes its models once every minute. The values measured in this experiment shows base resource utilization based on profile, location and sensor agents on the Android device.

As shown in Fig. 3.3, the average CPU utilization is lower at low battery levels. For the test, a 6.5 percent average CPU was observed at 95 percent battery level. At 40 percent battery level, average CPU used was 5.53 percent. With our profile settings, we also observe fewer network transmissions at a lower battery level as shown in Fig. 3.4. Our memory measurements don't show significant change in behaviour as no action agents are launched for this test as seen in Fig. 3.5. In this test, our current setting of the profile agent monitors the battery level of the Android device, and adjusts the frequency of data collected by the sensor and location agents. At lower battery levels, the profile and location update information is combined into a single ACL message. Thus, reduced traffic patterns is observed in Fig. 3.4.

3.5.2 GPS context test

This test was performed similar to the Battery Context Test. The primary observational difference however was that instead of measuring the CPU, memory or network utilization, as was done in the Battery Context Test, we observe which

Action agent is launched. Since Action agents are launched by the Profile agent, and die immediately after it performs its task, we log its output.

In order to test GPS context, we set our Profile Agent on the device to activate specific Action Agents only when we are within certain bounds of the Mock GPS. We create two zones for GPS tests. If the location agent reports GPS information in zone 1, the profile agent launches an Action agent named *zone1GPSActionAgent* which computes the values in Table 3.1. Otherwise it launches an Action agent called *zone2GPSActionAgent* that computes the same values. These computed values are sent to the Profile agent on the device using Agent Communication Language (ACL), which are then sent to the Profile Service agent on the server for computation. These feature values are passed to the relevant Device Analysis agents when launched to apply machine learning classifiers [5] using Weka [46].

This test run consists of the following steps:

1. Launch JADE agent platform on a network reachable server. This launches the JADE platform specific Directory Facilitator (DF) Agent and Agent Management System (AMS) agent; and the framework specific service agents: *Profile Service, Subscription Service, Location Service*.

2. An Android agent application is next launched on an Android Virtual Device. We provide the address of the network reachable server activated in the previous step to allow the Android Virtual Device to register itself with the Agent platform.

3. We then connect to the JADE service on the Android device and create the relevant data collection agents: *Profile Agent, Sensor Agent, Location Agent*. This launches a Device Analysis agent on the server.

4. We generate Mock GPS location events using a helper Android application. This causes the Location Agent to communicate with the Profile agent on the device and the Location Service agent on the server. This application generates location events every 1 to 5 seconds using a time-stamped location dataset. This is achieved using the following code snippet:

```
private Runnable mockgpsRunnable = new Runnable() {
public void run() {
Random r = new Random();
if (counter % 2 == 0) {
Intent i = new Intent();
i.setAction(MAIN_INTENT_ACTION_LOCATION);
i.putExtra(AGENT_LOCATION_LATITUDE, mock_lat);
i.putExtra(AGENT_LOCATION_LONGITUDE, mock_long);
getApplicationContext().sendBroadcast(i);
}
counter++;
TextviewLog("Mock location event generated");
mockgpsHandler.postDelayed(this,
1000 + r.nextInt(5000 - 1000 + 1));
}
};
```

5. We check the log to verify that the agent was started.

3.5.3 Network context test

This test was performed similar to the GPS Context Test. The primary observation was to check which Action agent is launched. Since Action agents are launched by the Profile agent, and die immediately after it performs its task, we log its output. In order to test Network context text, we set our Profile Agent on the device to activate specific Action Agents only when we are connected to certain wifi access points. We create two rules for Action agents. If the Sensor agent reports connection to wifi access point 1, the profile agent launches an Action agent named *zone1WifiActionAgent* which computes the values in Table 3.1. Otherwise it launches an Action agent called *zone2WifiActionAgent* that computes the same values. These computed values are sent to the Profile agent on the device using Agent Communication Language (ACL), which are then sent to the Profile Service agent on the server for computation as mentioned previously.

This test run consists of the following steps:

1. Launch JADE agent platform on a network reachable server. This launches the JADE platform specific Directory Facilitator (DF) Agent and Agent Management System (AMS) agent; and the framework specific service agents: *Profile Service, Subscription Service, Location Service*.

2. An Android agent application is next launched on an Android Virtual Device. We provide the address of the network reachable server activated in the previous step to allow the Android Virtual Device to register itself with the Agent platform.

3. We then connect to the JADE service on the Android device and create the relevant data collection agents: *Profile Agent, Sensor Agent, Location Agent*.

This launches a Device Analysis agent on the server.

4. We connect to the first wifi access point. This triggers the launching of Action agent zone1WifiActionAgent.

5. We check the log to verify that the agent was started.

6. We then connect to the second wifi access point. This triggers the launching of Action agent zone2WifiActionAgent.

7. We check the log to verify that the agent was started.

3.5.4 Android API context test

This test was performed to verify if Action agents only perform their tasks if they have the appropriate Android API level. As mentioned previously every Action agent maintains the Android API-level required for functionality. This is required as the functionality of the Agent might not be compatible with certain Android API-levels. For example, Android 4.3 allows applications to access Android notifications by creating a notifications listener. This feature was not available in earlier API levels. For this test, we incorrectly set an Action agent to be launched by the Profile agent on the Android device for an Android Virtual Device Version 4.0.3. If a check is not performed by the Action agent before launching a command, then it would fail with an exception *NoSuchMethodError*.

This test run consists of the following steps:

1. Launch JADE agent platform on a network reachable server. This launches the JADE platform specific Directory Facilitator (DF) Agent and Agent Management System (AMS) agent; and the framework specific service agents:

Profile Service, Subscription Service, Location Service.

2. An Android agent application is next launched on an Android Virtual Device set to 4.0.3. We provide the address of the network reachable server activated in the previous step to allow the Android Virtual Device to register itself with the Agent platform.

3. We then connect to the JADE service on the Android device and create the relevant data collection agents: *Profile Agent, Sensor Agent, Location Agent*. This launches a Device Analysis agent on the server.

4. After a time interval, we launch an Action agent that could only work without errors since version 4.3

5. We check the log to verify that the agent was started.

6. We check the log again to verify that the agent prints a message stating: API version of the Android device is not in range of the Action agent. We observe that no *NoSuchMethodError* exception was thrown.

3.5.5 Rooted device context test

This test was performed to verify if a rooted device can be successfully detected. In order to test this behaviour, we tested on a nexus-S Android device running in rooted mode. We also tested on a rooted Android Virtual Device, and a non-rooted Android Virtual Device. To perform this test, during the launch of Profile agent on the Android device, in its *setup()* method, we call a method in a helper class to check if the device is rooted. The helper class uses the RootTools program

mentioned previously. Following is a snippet of code that returns if the device is rooted:

```
import com.stericson.RootTools.RootTools;


public class JASHelpers {
/**
 * Check if root is available.
 * @return
 */
public static Boolean isRooted() {
return RootTools.isRootAvailable();
}
}
```

This test run consists of the following steps:

1. Launch JADE agent platform on a network reachable server. This launches the JADE platform specific Directory Facilitator (DF) Agent and Agent Management System (AMS) agent; and the framework specific service agents: *Profile Service, Subscription Service, Location Service*.

2. We provide the address of the network reachable server activated in the previous step to allow the Android Virtual Device to register itself with the Agent platform.

3. We then connect to the JADE service on the Android device and create the relevant data collection agents: *Profile Agent, Sensor Agent, Location Agent*.

This launches a Device Analysis agent on the server.

4. We check the log to verify that a message was printed by the Profile agent, stating the device root state of the device.

In all three instances of our test, the Profile agent successfully detected the correct root state of the device.

3.5.6 Malware detection test

This is a core functionality test of our agent-based malware detection framework. The purpose of this test was to verify if a feature vector that is known to exhibit malware is observed by our agent system, could it detect it.

In order to perform this test, we programmed an Action agent, that would report mock feature vectors known to be malicious from a previous training sample. i.e. We picked feature vectors from the training set that were marked as malicious, and were used to train the machine learning models.

This test run consists of the following steps:

1. Launch JADE agent platform on a network reachable server. This launches the JADE platform specific Directory Facilitator (DF) Agent and Agent Management System (AMS) agent; and the framework specific service agents: *Profile Service, Subscription Service, Location Service*.

2. An Android agent application is next launched on an Android Virtual Device. We provide the address of the network reachable server activated in the previous step to allow the Android Virtual Device to register itself with the Agent platform.

3. We then connect to the JADE service on the Android device and create the relevant data collection agents: *Profile Agent, Sensor Agent, Location Agent*. This launches a Device Analysis agent on the server.

4. The Profile agent launches the Action agent that has been programmed to return mock feature vectors.

5. The Profile agent receives the feature vector from the Action agent, and passes it to the Profile Service agent on the server.

6. The Action agent is destroyed as its task is complete.

7. The Profile Service agent forwards the feature vector to the Device Analysis agent.

8. The Device Analysis agent marks the feature vector as malware, and forwards the response to the Profile Service agent.

9. The Profile Service agent reports to the Profile agent on the device that it has exhibited malware behaviour at a certain timestamped value.

10. The Profile agent reports to the User Interface (UI) of the Android application that malware was detected.

11. The Profile agent writes to log the same information.

We observed the reported message in the UI of the Android device, and the log, confirming that the report is correct.

3.5.7 Verify Correlation agent can correlate 2 devices exhibiting similar malware behaviour

The purpose of this test is to verify if a Correlation agent can be used to connect multiple Android devices that exhibit malware on the same models. This test run consists of the following steps:

1. The pre-requisite for this test is that all steps of the Malware Detection Test described in the earlier section need to be done on 2 Android Virtual Devices.

2. We then launch another Android Virtual Device that does not exhibit malware behaviour.

3. We then launch a Correlation agent in a container.

4. The Correlation agent Queries the DF to find all Device Analysis agents in the system.

5. The Correlation agent sends queries to each Device Analysis agent to return it a list of models that have verified malicious behaviour and the set of feature vectors that caused it.

6. The Correlation agent then Correlates the information using the model as a key, a list of Android devices that failed the model, and the set of feature vectors that were sent to it when the model failed.

7. This information is stored in a JavaScript Object Notation (JSON) output file.

We observed a JSON output file created with the information about the two Android devices that did send mock malware feature vectors, and did not observe the third device listed in the JSON file.

3.5.8 Server workload measurement

In order to stress test the scalability of our solution on the server hosting the agent platform, we perform multiple experiments to measure the memory and CPU usage on the server.

Hardware

The server was run in a MacBook Pro 2012 with a 2.5 GHz Inter Core i5 processor and memory of 8GB DDR3 at 1600 MHz with a 5400rpm disk.

Memory measurements

In order to test the memory utilization on the server, we make measurements at each step of the server initiation. i.e. We make measurements every time a new Agent is launched in the main-container.

1. *When only the main-container is launched*: This measurement provides us the base resource utilization of a JADE system, without any framework specific agents launched. In this case by default a JADE Graphical User Interface (GUI) is launched with the AMS, DF and Remote Monitoring Agent (RMA) agents. The total memory reportedly used when the main-container is launched with these agents only is **91 MB**.

2. *When main-container is launched with framework specific agents*: This measurement provides us the base resource utilization when the JADE system is launched with framework specific service agents. In our case, the framework specific service agents are: Location Service, Profile Service and Subscription Service. The total memory reportedly used when the main-container is

launched with a GUI and framework specific agents is **105 MB**. Thus, the overhead for the three framework specific agents is **14 MB**.

3. *When Device Analysis agent is launched*: This measurement provides us the resource utilization of the server when an Android device joins the framework, thus causing the Profile Service agent to launch a Device Analysis agent on the server. At this measurement instance, the agents on the server include: AMS, DF, RMA, Location Service, Profile Service, Subscription Service and a Device Analysis agent. The total memory reportedly used in this instance is **111 MB**. When a second Device Analysis agent is launched, the total memory usage is **118 MB**. Thus, a Device Analysis agent has an overhead of **7 MB**. Note that at this instance, no machine learning models have been loaded by the Device Analysis agent. The models are loaded only when one or more feature vectors are received by the Device Analysis agent from the Profile Service.

4. *When Device Analysis agent is loaded with models*: This measurement provides us the resource utilization of the server when a feature vector is received by the Device Analysis agent, causing it to load pre-existing machine learning models to compare against. At this measurement instance, the agents on the server include: AMS, DF, RMA, Location Service, Profile Service, Subscription Service and a Device Analysis agent. Moreover the Device Analysis agent loads **1350** preexisting models. Each model ranges in size from **76 KB** to **15.2 MB** depending on the complexity of the model. The total memory utilization reported is **559 MB**. Thus, **448MB** extra is used to load preexisting models per Device Analysis agent in our case. This value

shows that if there are many models to compare against, or the size of each model is large, the number of Device Analysis agents launched in a server have to be reduced.

CPU usage time measurements

Most of the agents on the server side have limited CPU utilization. The only agent that consumes large amounts of CPU is the Device Analysis agent. This is because the Device Analyis agent is the one that has to load machine learning models and compare the feature vectors received with each of the models available. Based on [22], as the number of trees in the model increase, the CPU time required will increase linearly. Similarly, if the depth of trees in the model increases, the CPU time required will increase. Here we compare the CPU utilization time in different scenarios when the Device Analysis agent is run. We run each test 3 times to reduce the variance in times between test runs. For all tests, the Device Analysis agent compares feature vectors with **1350** stored machine learning models. Unless stated otherwise, the number of feature vectors compared by the Device Analysis agent against each model is **256**.

1. *When only 1 Device Analysis agent runs*: In this case, there exists only 1 Device Analysis agent in the system. The CPU usage times reported for three test runs were: 27.4 seconds, 27.6 seconds, 28.9 seconds. Thus, we see an average CPU utilization of **28.0** seconds.

2. *When 2 Device Analysis agents run simultaneously*: In this case, there exists 2 Device Analysis agents, that have received 256 feature vectors each, to compare against 1350 stored machine learning models. The CPU usage

times reported by the first Device Analysis agent for three test runs were: 33.4 seconds, 32.4 seconds, 33.6 seconds. The CPU usage times reported by the second Device Analysis agent for three test runs were: 32.6 seconds, 32.3 seconds, 33.6 seconds. Thus we see an average CPU utilization of 33.0 seconds. This shows that the CPU utilization time increased by **5.0** seconds when another Device Analysis agent is introduced in the same server.

3. *When only 1 Device Analysis agent runs with 1 feature vector comparison*: In this test, we measure the CPU utilization overhead when only 1 feature vector needs to be compared by the Device Analysis agent. In previous instances, the Device Analysis agent had to compare 256 feature vectors. The CPU usage times reported by the Device Analysis agent for three test runs in this case were: 25.2 seconds, 22.6 seconds, 25.0 seconds. Thus, we see an average CPU utilization of **24.3** seconds. This is **3.7** seconds quicker than comparing 256 feature vectors. This shows that though a single feature vector can be processed faster, the time required per feature processing is reduced if more feature vectors are compared at a time together.

3.6 Discussion

Based on our experimental results, we make the following observations:

- **Server memory should be high** If the number of machine learning models to be loaded is high, based on the amount of memory required by Device Analysis agents either the Device Analysis agents need to be distributed among multiple servers, or the memory used to run the agent framework needs to be high. In our experimentation loading the models caused a 448

MB memory overhead. Each server-side agent itself uses between 5 - 7 MB of memory.

- **Pre-loaded machine learning models can give quick results** Based on measuring the CPU time required to compute if a mobile device exhibits malware, we can say that multiple models can be loaded quickly, and classification results obtained. We observed that it took 24.3 seconds to compare one feature vector against 1350 stored machine learning models; and 28.0 seconds to batch-process 256 feature vectors against 1350 stored models.

- **Context-based decisions** In our experimentation, our framework made changes to its monitoring behaviour based on battery availability context, network context, GPS context, Android API context, and rooted device context. For example, we reduced the CPU and network usage when the available battery was below a certain threshold level. As mentioned previously, the context-based decisions depend on having access to sensors and appropriate API to read the state of the device. With the changing nature of the Android operating system, either more context-based behaviours could be modeled or fewer obtainable if the system becomes more closed.

- **Agent-based approach is viable** In our experimentation, our framework used agents on the Android device in a split-container execution mode, communication with agents on the server side for analysis. The communication between the client and server was done using Agent Communication Language. Given that some of our agents are resource intensive, such as the Device Analysis agents, the agent-based approach allowed us to launch only the minimum data gathering agents on the client, while allowing the resource

intensive server agents to handle storage and computations using machine learning models.

- **Device infections can be correlated based on model failure** In our experimentation, Correlation agents can be used to query Device Analysis agents to detect devices that fail on the same models. This is done by first querying the Directory Facilitator agent service to find all Device Analysis agents in the system, and then querying each Agent to provide its saved results. This result is saved in a JSON file for future reference.

Some of the concerns that could be considered as issues or weaknesses of the proposed agent-based framework includes:

- **Privacy** The information collected by the agents in the system cover a wide range of features such as GPS position information; network access points connected to; application installations; memory and CPU usage monitoring; IP addresses and possible packet inspection if a device is rooted. This could raise privacy concerns if our current solution is to be widely acceptable. It should be mentioned that the current Android API allows access to most of this information to all installed applications on the device without having to root the device. For example, the *Network Connections* applications by Anti Spy Mobile [14] allows capturing the IP addresses and ports being communicated with; the application package causing the communication; amount of data transferred and the Transmission Control Protocol (TCP) state of the communication. All this information can be accessed without having to root the device. Moreover, Android requires any access permissions required by

our application to be accepted by the user before the application can be installed on the device. Thus, users are well-aware of what kind of information can be accessed.

- **Agent-based client-server communication** As mentioned previously, the current system architecture is a client-server model with most analysis services residing at the server side. This would mean that no malware can be classified while the device cannot connect to the server. Use of the current JADE agent-based architecture allows asynchronous communication during client-server disconnections. This is made possible by the split-container execution mode in JADE supported by the LEAP add-on. In case of connection loss, feedback is stored in the Android devices. Once a connection is re-established, the relevant data gathered during the disconnection time is passed to the agent on the server side with appropriate time stamped data. As mentioned previously, one of the advantages of an agent-based system is asynchronous communication. Moreover, as mentioned in section 3.3.3 use of Agent Communication Language allows us to distribute the agents on the server side on multiple physical machines for scalability.

3.7 Summary

To summarize, in this chapter we provided an agent-based framework based on the JADE middleware to make context-aware decisions to detect Android malware. The contexts that we used included battery context, network context, GPS context, Android API level context and rooted-device context. We conducted multiple experiments based on these contexts, and measured the client and server workload

measurements in terms of CPU and memory utilization. We also performed correlation experiments to detect if mobile devices that fail on the same models can be correlated together.

Chapter 4

Random Forest Classification For Detecting Android Malware

4.1 Introduction

The purpose of this chapter was to demonstrate the feasibility of using the Random Forest classification algorithm to detect if an Android device has been compromised by malware by inspecting application behaviour data. As mentioned in the previous chapter, the Device Analysis agents in our framework use pre-built machine learning classification models to classify devices as exhibiting malware behaviour. The classification model generated is based on the Random Forest algorithm, as described in this chapter.

There were 3 primary reasons for choosing the Random Forest classification algorithm over other machine learning algorithms. They were:

1. No previous research work had verified the classification performance of the algorithm on an Android dataset by modifying the parameters of the

algorithm.

2. As shown in Table 4.5, our initial investigation showed that the algorithm already performed better than other machine learning classifiers on our dataset.

3. As will be described in Section 4.2.2, the algorithm is highly scalable as it required a log scale comparison of available features to generate a classification model. This allows models to be generated quickly from large datasets.

In this chapter we focus on training a classifier based on an offline analysis of Android application behaviour. Once trained, this technique could then be used to analyze new Android applications prior to deployment on a real Android device. This could be done by testing computed feature vectors collected during run-time of the application on an Android Virtual Device.

For experimentation, we used a modified dataset made available by Amos [9] under the Creative Commons Attribution 3.0 Unported License downloaded in Feb 28, 2013. This dataset was obtained from emulating user action using adb-monkey [44]. More information regarding the dataset is provided in Section 4.3.1. We focus on the detection accuracy of Random Forest as the number of trees, depth of each tree and number of random features selected are varied for the algorithm. We perform a 10-fold cross validation on our dataset for the error estimate, and also use a separate training and validation set test. We had also performed a 5-fold cross validation experiment for which an optimal Out-Of-Bag (OOB) error rate [22] of 0.0002 for forests with 40 trees or more, and a root mean squared error of 0.0171 for 160 trees was obtained.

According to our knowledge, we have not come across any other related work that has exclusively worked with the free parameters of Random Forest algorithm

on an Android feature dataset for malware classification. The main contributions of this chapter can be summarized as follows:

- Apply Random Forest classification algorithm for behavior detection on features [9] obtainable from an unrooted Android devices by predominantly monitoring features based on the Android Binder Application Programming Interface (API), Memory and Central Processing Unit (CPU) measurements to detect Android malware.

- Conduct detailed experimentation to measure the accuracy of Random Forest classifier as the number of trees and the depth of each tree in the forest is varied for different size of random features selected.

- We provide a 10-fold cross validation of results of application of the Random Forest classifier on android applications to detect malware. For this we conduct 3 test runs for each combination of the Random Forest parameters by using 3 random seeds for each of our tests to provide adequate error measurements. We test for the number of trees, number of features compared at a tree node, and the depth of the tree. For the number of trees, we test with 10, 20, 40, 80 and 160 Trees. For features we compare 4, 6, 8, 16 and 32 features. For tree depth we compare 1 (decision stump), 2, 4, 8, 16, and 32 depth trees.

- We perform similar test to see how a Random Forest training model performs if there is a separate training set and a separate validation set. We compare the performance of the original dataset, and our modified data set to measure performance trends in terms of using a validation set.

- We also perform a comparison of 10-fold cross validation by the usage of SMOTE [26] (Synthetic feature vector generation using the default settings of Weka) as the publicly available set did not have enough samples of the benign applications. There was a 2:1 ratio of malware to benign samples.

- We also measure the amount of space required in storing different Random Forest models on disk. This is required as once models are precomputed, they can be used to evaluate incoming feature vectors.

The rest of the sections are organized as follows. Section 4.2 provides relevant background to understand the problem domain. In Section 4.3 we describe our experimental approach. Section 4.4 discusses our results, the advantages and disadvantages.

4.2 Background

In order to understand the application domain of this research work, we now discuss three related areas: feature collection for Android operating system, key features of the Random Forest algorithm, and application of machine learning in the domain of mobile security.

4.2.1 Android feature collection

In order to apply any machine learning classifier, it is important to first be able to collect relevant features that can be observed from the system. These features that are observed are stored as a feature vector. As mentioned in Section 2.3, in supervised learning, given a labeled set of input-output pairs (the training set) with N training examples, the goal is to predict a future y value given an input. The

input to such a system is x_i which is a D-dimensional vector, also called a feature vector. x_i is a set of values used to determine y_i.

The features that the Android system allows access permissions to depends on if the device has been rooted. Rooting is the process whereby one has privileged access control to the system. As mentioned in Android Security Overview [42], Android uses the Linux Kernel as the bottom layer. All layers on top of the kernel layer run without root privilege. i.e. All applications and system libraries are inside a virtual application sandbox. Thus, applications are prohibited from accessing other application data (unless explicitly granted permission by other applications). As described in [42], each Android application is assigned a unique User Identification (UID). Each of these Android applications run as that user in a separate process with the UID. Thus, if a feature vector is created from features of Android API in unrooted mode, then only system information made available by Android can be used. On the other hand, having a rooted device allows one to install system tools that could gather features from underlying host and network behaviour. Example of features that can be obtained from rooted devices include: data being sent by applications, IP addresses being communicated with in the network, number of active connections, the system calls being invoked, etc. The related work in the area of machine learning approaches to Android data either deal with feature collection from unrooted devices [9, 71], or rooted devices [24, 34, 49]. Hence, if models are to be generated with root mode observable features, then root access is required. If models are to be generated without root mode features, then root access is not required.

4.2.2 Random forest

For the experimentation in this chapter, we have used the Random Forest classification algorithm implemented in the standard Weka [46] package. Weka is the only Java-based package available that provides implementation of all machine learning algorithms, including the Random Forest classification algorithm. As our agent-based system is based on the Java language, and as Android is a derivative of Java, we have used the Weka package. Random Forest is an ensemble learning algorithm developed by Breiman [22]. An ensemble learner method generates many individual learners (in this case a tree) and aggregates the results (a forest). Each classifier (a tree in the forest) is built individually by working with a bootstrap sample (random sampling with replacement) of the input data (features of the tree). In a regular decision tree classifier, a decision at a decision point is made based on all the feature attributes of a feature vector. But in Random Forest, the best parameter at each node in a decision tree is chosen from a randomly selected number of features. In our experimentation we choose 4, 6, 8, 16 and 32 features to compare at a decision point. The feature that is chosen to classify the data is based upon trying to minimize the entropy value. This random selection of features helps Random Forest to scale well when there exists many features per feature vector as the algorithm usually requires as few as log m features being compared at the classification decision point of a tree, where m represents the total number of features available in the feature vector. This feature selection helps it to reduce the interdependence (correlation) between the feature attributes. The error rate of the forest classification depends on the optimal value of random features selected.

As mentioned by the author [22], the number of random features m selected

per decision node in a tree determines the error rate of the forest classification. The error rate of the Random Forest classifier depends on the *correlation* between any two trees, and the classification *strength* of each individual tree. Reducing the random features selected *m* causes reduction in both the correlation between classification trees and the strength of classification of each individual tree. Increasing *m* increases both the correlation between the trees and the strength of each tree.

Breiman explains that the Out-of-Bag (OOB) error rate is an indication of how well a forest classifier performs on the dataset. The out-of-bag model leaves out one-third of the input dataset for building the *k*th tree from the bootstrap sample for each tree. This one-third sample is used to test the *k*th tree and the results of misclassification averaged over all trees. The author claims that for most cases OOB error estimate is a good estimate of the error and hence cross validation or separate test set is usually unnecessary when using the Random Forest algorithm [23].

4.2.3 Machine learning classifier for Android mobile systems

Application of machine learning methodologies for classification of Android malware is currently an emerging area of interest. Malware detection can be done either via static detection or dynamic detection. Static detection techniques (also called signature matching) have high detection rates and require less computational resources. This is the traditional approach taken by anti-virus software. Dynamic detection techniques (also called anomaly/behavior detection) on the other hand, suffer from low detection rates and require more resources to reduce the amount of false positives reported. They usually are prone to high false positives without adequate training. The primary advantage of dynamic detection techniques over

static detection techniques is however that it can detect new malware, and provides better coverage. A description of machine learning classification used on Android dataset was provided in Section 2.3.1.

4.3 Experiment

4.3.1 Dataset description

For our experimentation, we worked with a modified dataset provided by Amos et al. in [10] available at [1] under the Creative Commons Attribution 3.0 Unported License. The author developed a shell script to automatically analyze .apk Android application files by running them in available Android emulators. For each .apk file, the emulator simulates user interaction by randomly interacting with the application interface. This is done using the Android adb-monkey tool [44]. Each feature vector of the dataset is collected at 5 second intervals. The memory features were collected by observing the"proc" directory in the underlying Linux system of Android. The CPU information was collected by running the Linux "top" command. The Battery and Binder information was collected by using an "intent" object. The "permissions" feature is the summation of the total permissions requested by each running application (package) obtained from the *PackageManager* class in Android.

As mentioned in [10], the test set or the validation set consists of 47 applications of which 24 are benign applications and 23 are malicious applications. The training dataset consists of 6832 feature vectors computed. Eight applications are duplicated across both the sets. The feature vectors in the training set were com-

[1] https://github.com/VT-Magnum-Research/antimalware

puted from 1330 malicious applications and 408 benign applications. As can be seen, there is an imbalance in the number of applications observed in the training set. A classifier becomes more biased towards the oversampled class if there is a class imbalance. In order to balance the two classes, we used the Synthetic Minority Oversampling Technique (SMOTE) package from Weka to create more data points of the under-sampled class (benign class). We then applied the random filter to randomize the distribution of the two classes in the data file used. Randomization was required as SMOTE adds data entries to the end of the data file. The synthetically generated data without randomization would be a problem during cross validation because we would have points primarily from the under sampled class in the last few folds.

For the SMOTE tests, we generated more samples of the under sampled benign class to cause the total number of feature vectors to increase from 6832 to 10254 feature vectors. 5133 instances were of one class whereas 5121 were of the other class. In our initial experiments reported in [5], we used a data set collected from [2] that had a total of 32342 data (feature vector) samples with 7535 benign samples (classified as positive class) and 24807 malicious samples (classified as negative class). A reduced version of the dataset was reported in [10] which has been used by us now. For completeness, we report our results in [5] in section 4.3.7.

The feature vectors were computed every 5 seconds from 10,000 user inputs generated by the Android Monkey program. The primary disadvantage of the dataset however as mentioned by the authors is that when they worked with 1500 events that were sent with random delays, their classifier performed only as good as random behaviour with around a 50 percent classification rate. The primary fo-

[2]https://github.com/VT-Magnum-Research/antimalware

cus of [10] was to create an architecture for large amount of testing using the base settings of machine learning classifiers. They tested the Random Forest algorithm using just 10 trees, 6 features, with maximum tree depth. We tested their unbalanced dataset, and compared with our balanced SMOTE generated dataset for the various parameter settings of the Random Forest algorithm. We test by modifying the following parameters: The number of trees, number of features compared at a classification decision, and the depth of the tree. For the number of trees, we test with 10, 20, 40, 80 and 160 Trees. For features we compared with: choosing the best among 4 features, choosing the best among 6 features, choosing the best among 8 features, choosing the best among 16 features and choosing the best among 32 features. For tree depth we compare trees of depth: 1 (decision stump), 2, 4, 8, 16, and 32. We also run each experiment three times and report the median value for each combination of the tests. We run a total of (5 trees * 5 features * 6 depth * 3 random seed) = 450 tests for each of our test setups. We run 4 set of tests. 1) 10-fold Cross Validation on original dataset, 2) 10-fold Cross Validation on our Synthetic data test (SMOTE generated), 3) Separate Validation set test on the original data, 4) Separate Validation test on our Synthetic data test (SMOTE generated). Thus, we run a total of 1800 tests. The features that are measured in the feature vector are listed in Table 4.1. This is the same set of features that were described in Table 3.1. They are shown again here for easier reference.

This study has two additional limitations that we observed based on inspecting the raw dataset. The most notable areas are in the battery data features and the network features. Given that these data samples were collected by running in an emulator, there were no observed change in battery data among any of the feature vectors for both classes of data. Similarly, the Network data was fixed for all the

Table 4.1: Android Features Observed [5]

Category	Feature
Battery	IsCharging, Voltage, Temperature, BatteryLevel, BatteryLevelDiff
Binder	Transaction, Reply, Acquire, Release, ActiveNodes, TotalNodes, ActiveRef,TotalRef, ActiveDeath, TotalDeath, ActiveTransaction, TotalTransaction,ActiveTransactionComplete, TotalTransactionComplete, TotalNodesDiff, TotalRefDiff, TotalDeathDiff, TotalTransactionDiff, TotalTransactionCompleteDiff
CPU	CPU Usage
Memory	memActive,memInactive, memMapped, memFreePages, memAnonPages, memFilePages, memDirtyPages, memWritebackPages
Network	TotalTXPackets,TotalTXBytes, TotalRXPackets TXPacketsDiff, TXBytesDiff, RXPacketsDiff, RXBytesDiff
Permission	Permission

data points. The authors reported in [10], that the network was switched off during tests due to complaints by the information technology team. This would mean that 13 of the observed 42 features do not contribute to classification as the 13 features will not contribute to information gain (reduction in entropy) score used by Random Forest algorithm at each decision node. These limitations could only be addressed if the network is allowed to remain open so that Android applications can access network data, and actual mobile devices are used instead of Android virtual devices. But running 1738 Android applications, as was done to gather this dataset, would be time consuming on actual devices as it would require device software reset after each installation of Android application.

4.3.2 Classification experiment description

The following are a list of experiments performed:

- *Experiment 1*: Use initial dataset to train classifier on the training dataset provided in [10]. Perform a 10-fold cross validation of results. We perform 3 runs with seed values, 1, 2 and 3 to measure the performance with multiple settings of the Random Forest classifier as described above.

- *Experiment 2*: Next, again train the Random Forest classifier on Amos et al. [10] training dataset, but now check the performance of each Random Forest model on a separate validation set. As mentioned previously, the test set or the validation set consists of 47 applications of which 24 are benign applications and 23 are malicious applications.

- *Data Generation*: We then modified [10] training dataset using the SMOTE package of Weka to generate synthetic feature vectors for the training set. This was done as the number of feature vectors of goodware was significantly low. This causes the learned classifiers to be over trained on the class that is oversampled. We applied the randomize filter then since the SMOTE package adds the synthetic features to the tail of the Weka .arff file. This would create errors during the process of cross-validation.

- *Experiment 3*: We then performed a 10-fold cross validation on the balanced dataset generated using SMOTE. We perform 3 runs with seed values, 1, 2 and 3 to measure the performance with the settings of the Random Forest classifier we mentioned previously. This is similar to what was done in experiment 1, and would help us compare the effect of a balanced dataset.

- *Experiment 4*: Next, again train the Random Forest classifier on the balanced dataset, but now check the performance of each Random Forest model on the separate validation set, as was done in Experiment 2. This would allow us to compare the performance of the two models: One generated from the original dataset, to the one generated from synthetic oversampling.

4.3.3 Hardware

A Macbook Pro 2012 was used for testing with a 2.5 GHz Intel Core i5 processor with 8 GB 1600 MHz DDR3 of available RAM.

4.3.4 Software

For generating synthetic data for solving the class imbalance issue, we used SMOTE (Synthetic Minority Oversampling Technique) [26] package available as part of Weka 3-6-12. For experimentation and model generation, we used the experimental release 3-7-12 of Weka as it allows parallel processing of classifier for Model generation for the Random Forest algorithm. We set the number of threads available to be used by the Random Forest algorithm to four as that is the limitation of our current hardware.

The Random Forest classifier implementation for Weka does not implement feature importance computation. Availability of this feature would have helped in observing the features that are weighted higher by Random Forest algorithm. Hence, we have not computed feature importance.

4.3.5 Sample code

As mentioned previously, in order to be able to compute if a feature vector exhibits maliciousness, we have to use pre-computed Random Forest models stored in the

system. To do this, we write java code to store models once they have been generated. This is done using serialization support available in Weka. Below is our sample on how this is performed.

```
rfa.buildClassifier(data);
// serialize model
weka.core.SerializationHelper.write(
   inputFileName+"RandomForest"+
   rfa.getNumTrees()+"Trees"+
   rfa.getNumFeatures()+"Features"+
   rfa.getMaxDepth()+"Depth"+
   rfa.getSeed()+"Seed.model", rfa);
```

In order to load a classifier model already created and stored on disk, we use the following java code to deserialize.

```
// retrive the model file to test with
Classifier cls = (Classifier) weka.core.
      SerializationHelper.read(modelFile);
// Evaluate
eTest = new Evaluation(data);
eTest.evaluateModel(cls, data);
```

Once loaded, the new incoming feature vectors are compared using the data variable. Weka provides the necessary infrastructure to either load multiple feature vectors to test at a given time, or a single feature vector can be compared at a time.

In our code, the incoming features stored in the data variable are collected on an actual or emulated Android device. The classification is done on a server side.

If required, the model files can be stored on an Android device as Weka is supported on an ARM architecture as well, and the comparison can be done on the Android device. It should be mentioned that building the classifier is a CPU intensive task requiring tens of minutes even on a multicore desktop or laptop for the current data set that we have. The evaluation however can be done on a handheld device as they are memory intensive, but not CPU intensive tasks. As explained in our previous chapter, it took 28 seconds to compare against 1350 models.

4.3.6 Results

As mentioned before, for experimentation, we run tests for each of the Number of Trees , Depth of Tree and Attribute combination. We tested with 10, 20, 40, 80 and 160 trees for each Random Forest. For each setting of trees we tested with 4, 6, 8, 16 and 32 randomly selected attributes compared for each decision node. For each Tree and attribute combination, we tested with 1, 2, 4, 8, 16 and 32 depth trees.

Fig. 4.1 shows the 10 fold cross validation performance of the original dataset. As can be seen, as the depth of tree increases, the classification rate increases up to a certain point. The range shown is from 80 percent to 96 percent on the y-axis. The highest correct classification detection rate was a value of 94.96 percent for 16 features at 32 depth for a Random Forest of 10 trees.

As shown in Fig. 4.2, a similar trend follows if we use more trees compared to Fig. 4.1. However, though it is not noticable, the detection rate increases slightly by one to three percent for 10 trees vs 160 trees. For 160 trees, the highest correct classification detection rate was a value of 95.74 achieved with 32 features

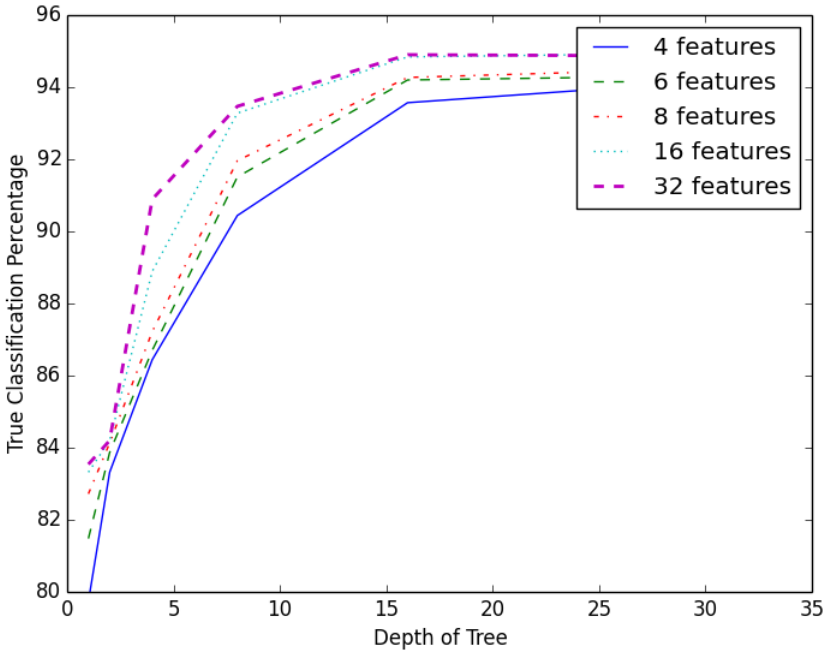

Figure 4.1: Classification rate of original dataset as the number of features are changed for forest of 10 trees for y range 80 to 96 percent

compared at a depth of 32.

In Fig. 4.3 we compare the performance of the various Random Forest models on the original dataset where we have a separate training set and a separate validation set. As can be seen, the performance of the test on a forest of 10 trees on a validation set is much lower than in the 10-fold cross validation test for 10 Trees. This shows that a 10-fold cross validation set does not always show the true nature of a classification algorithm. The highest detection rate observed was 80.46 for 16 features compared at a time with tree depth of 4. Moreover, the results show that as the depth of tree increases, the classification results are poorer due to over-fitting.

We performed a similar test with a 160 tree forest, and found similar results

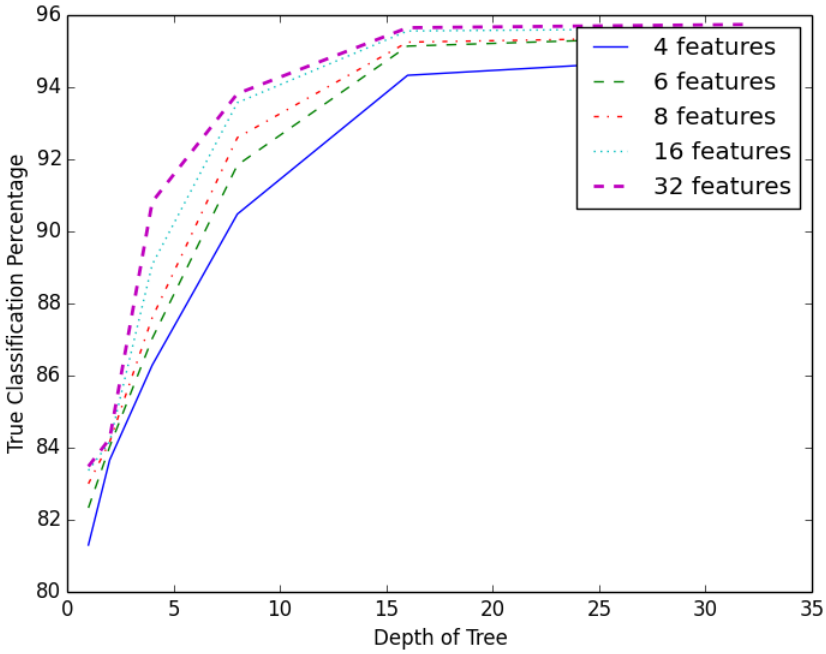

Figure 4.2: Classification rate of original dataset as the number of features are changed for forest of 160 trees for y range 80 to 96 percent

to the 10-Tree validation set. The results are shown in Fig. 4.4. The highest detection rate was a value of 81.25 observed with 4-feature comparison at a depth of 8. Moreover, with 160 trees, there were a total of 3 tests that passed 80 percent detection. There was only one such case with 10 Trees.

As shown in Fig. 4.5, we performed a test where the model was generated on a separate training set, and then tested on a separate test set as mentioned in our dataset description. In this test we had synthetically generated feature vectors of the under sampled class using the SMOTE algorithm in its default setting. The highest correct classification rate achieved was 81.64 percent with a setting of 160 trees, depth of 2, with 16 random features at a time. The performance of SMOTE

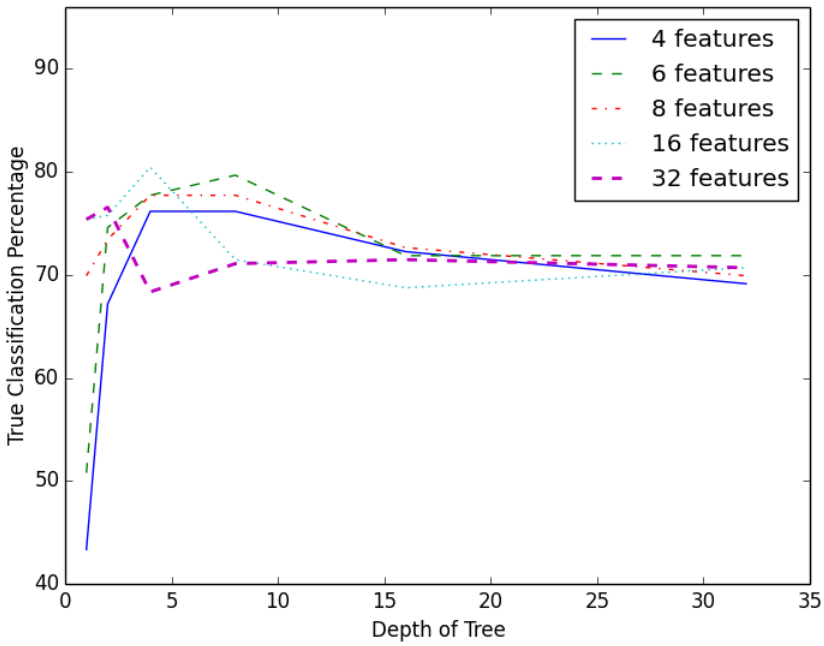

Figure 4.3: Classification rate of original dataset when tested on validation set for forests of 10 trees. The results show that as the depth of tree increases, the classification results are poorer due to over-fitting.

can be compared to the original dataset of Amos for 160 Trees on a validation set as was done in Fig. 4.4. There were 6 tests where the correct classification passed the 80 percent detection. They were all achieved at a depth of 2 for 4, 6, 8, 16, 32 features. However, the original dataset performed better when the tree depth was 4 or greater in general.

As shown in Fig. 4.6, we also performed a 10 fold cross validation test on SMOTE generated instances. The detection rate was over 91 percent for tree depth of 8 or more. The highest classification rate obtained was a value of 96.40 percent for trees with 8 features compared at 32 depth.

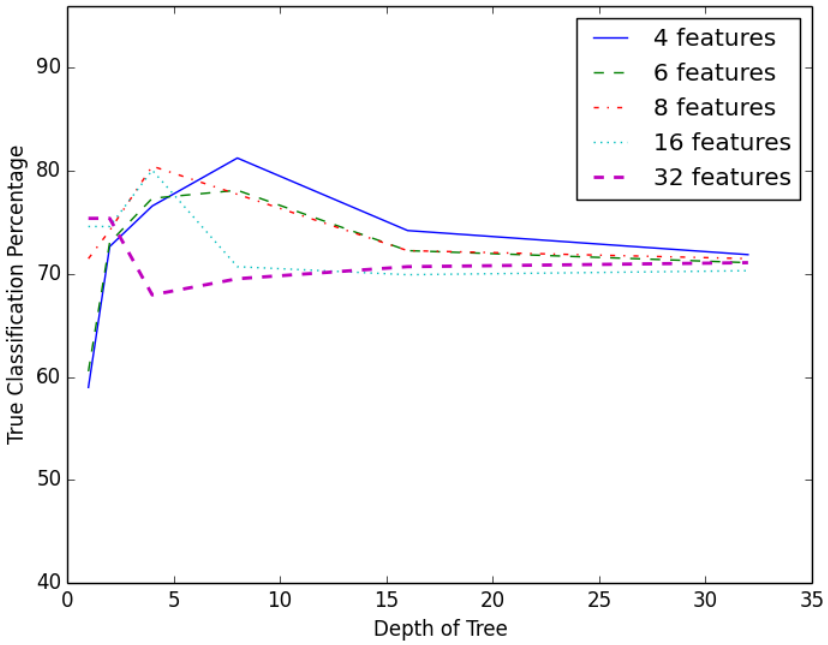

Figure 4.4: Classification rate of original dataset when tested on validation set for forests of 160 trees

Table 4.2: Amount of space required to store Random Forest model files as depth of trees is increased for a forest of 160 Trees

Trees	Features	Depth	Model Size(MB)
160	32	32	7.9
160	32	16	7.5
160	32	8	3.4
160	32	4	1.9
160	32	2	1.7
160	32	1	1.7

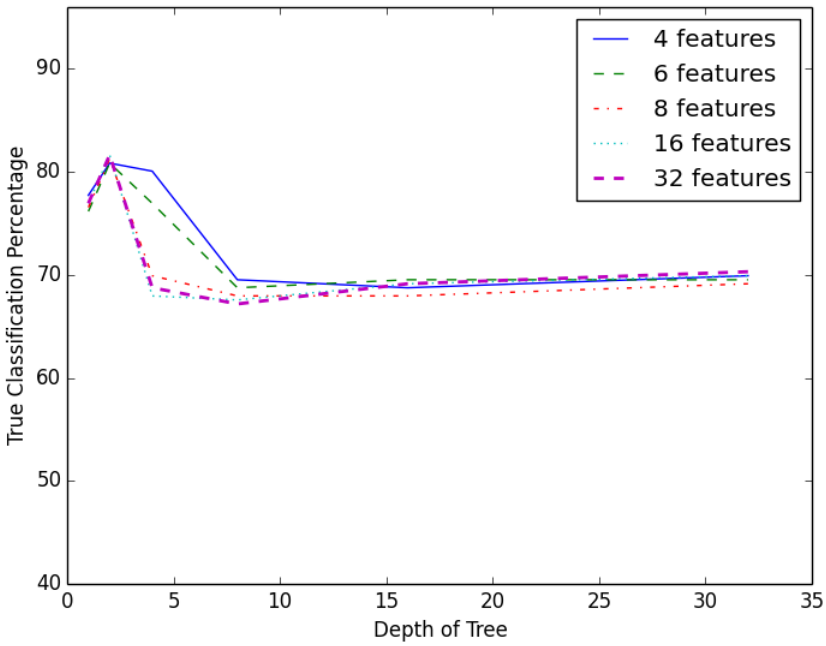

Figure 4.5: Classification rate on model generated by SMOTE with testing on a validation set with forest of 160 trees. This graph shows the performance of the Random Forest algorithm on a balanced dataset.

In order to examine if storage of Random Forest models are feasible on mobile devices, we measured the amount of space occupied by such models. This is shown in Table 4.2. As can be seen, the amount of space required increases as the depth of the tree increases. Random Forest models with 160 Trees, 32 Features compared at a time (with replacement), with tree depth of 16 would require 7.5 Megabytes of storage.

As shown in Table 4.3, the amount of space required to store a Random Forest model decreases as the number of features compared increases.

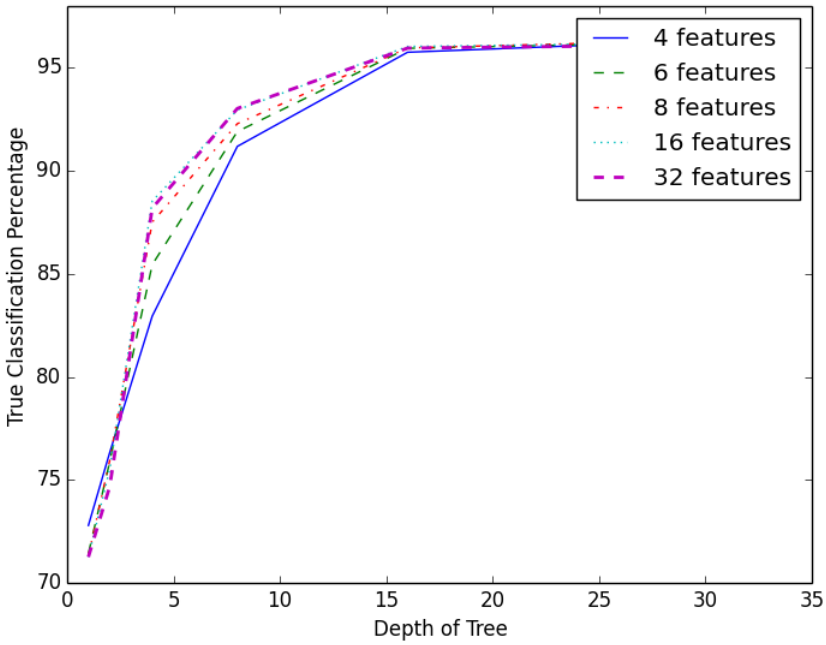

Figure 4.6: Classification rate on model generated by SMOTE with 10 Fold cross validation testing with forest of 160 trees

Table 4.3: Amount of space required to store Random Forest model files as number of features measured is increased for a forest of 160 Trees

Trees	Features	Depth	Model Size(MB)
160	4	32	15.1
160	6	32	12.7
160	8	32	11.4
160	16	32	9.2
160	32	32	7.9

4.3.7 Experiment results with a 5-fold cross validation

As mentioned previously, we had also performed a 5-fold cross validation experiment in our previous work [5] based on a different dataset provided by the authors. Using a lower number of folds (using 5 instead of using 10) means a higher bias towards overestimating the true expected error. This however causes lower variance and lower running time. The following were the results during our 5-fold experimentation. The results here are based on applying the SMOTE algorithm to remove class imbalance on the original dataset.

Table 4.4 shows the results when each tree is allowed to grow to maximum depth. We performed 5-fold cross validation for each experiment.

Following are the parameters that we measured as shown in Table 4.4:

- *OOB Error*

  This is the Out-Of-Bag error explained in section 4.2.2.

- *Root MSE*

  The square root of the mean squared error based on 5-fold cross validation.

- *% True Class*

  This value shows the percentage of the samples that were classified correctly. We evaluate this number to show the minor variations that happen as the number of trees and random features selected change. This was not reported in 10-fold cross validation as the variation is more obvious.

- *# Incorrect*

  This number shows the total number of the 48,919 samples that were misclassified. We evaluate this number to show the minor variations that happen

Table 4.4: Experimental Results for varying number of trees and number of random features with Trees allowed to grow to maximum depth

Trees	Features	OOB Err	Root MSE	% True Class	#Incorrect
10	4	0.0067	0.0291	99.9469	26
10	6	0.0064	0.0245	99.9407	29
10	8	0.0058	0.0221	99.9600	19
10	16	0.0056	0.0206	99.9600	19
10	32	0.0056	0.0200	99.9600	19
20	4	0.0008	0.0259	99.9670	16
20	6	0.0006	0.0223	99.9632	18
20	8	0.0005	0.0208	99.9693	15
20	16	0.0004	0.0191	99.9652	17
20	32	0.0006	0.0184	99.9693	15
40	4	0.0003	0.0242	99.9693	15
40	6	0.0004	0.0210	99.9734	13
40	8	0.0002	0.0197	99.9755	12
40	16	0.0003	0.0178	99.9612	19
40	32	0.0003	0.0178	99.9714	14
80	4	0.0004	0.0239	99.9734	13
80	6	0.0002	0.0203	99.9775	11
80	8	0.0002	0.0187	99.9836	8
80	16	0.0002	0.0175	99.9734	13
80	32	0.0003	0.0178	99.9673	16
160	4	0.0003	0.0233	99.9775	11
160	6	0.0002	0.0201	99.9755	12
160	8	0.0002	0.0183	99.9857	**7**
160	16	0.0002	**0.0171**	99.9734	13
160	32	0.0002	0.0175	99.9693	14

as the number of trees and random features selected change.

As shown in Table 4.4, the best Root Mean Squared Error value of 0.0171 is achieved with 160 trees and 16 features. The best setting based on the lowest number of misclassifications (7 out of 48,919) was achieved with 160 trees and 8 features selected. These have been highlighted in Table 4.4.

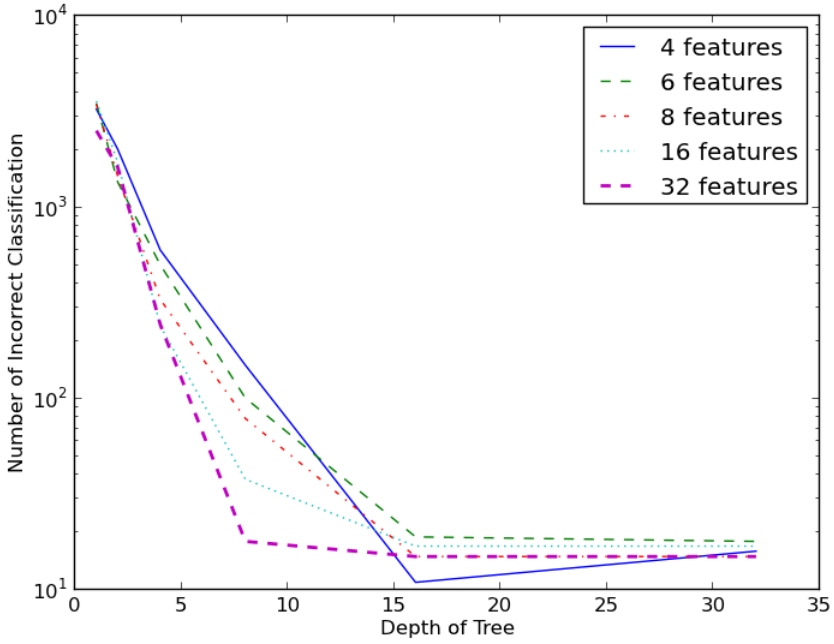

Figure 4.7: Misclassification comparison with 20 trees as depth of tree is varied

Fig. 4.7 provides comparison of how the depth of tree impacts the number of inaccurate classifications. In this figure we show results of a Random Forest with 20 trees. As can be seen, regardless of the number of features selected, the number of incorrect classifications stabilizes at a tree depth of 16. Similar behavior is observed for other size of forests, i.e. forests with 10, 40, 80 and 160 trees. We omit them here for brevity.

Fig. 4.8 provides comparison of how the depth of the tree impacts the out-of-bag error rate for Random Forest with 40 trees. As can be seen from the figure, low error rates are achieved quickly if 16 and 32 features are selected at tree depth of 8. If lower number of features are selected, then a tree depth of 16 is required

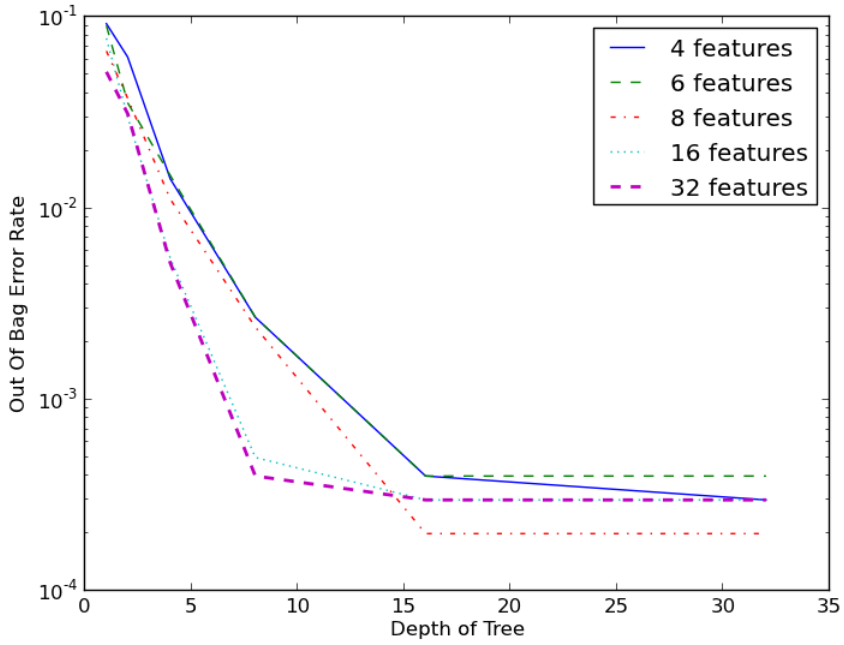

Figure 4.8: Out Of Bag error rate comparison with 40 trees as depth of tree is varied

before the error rate stabilizes. Similar behavior is observed for other forest sizes for the given tree depth and number of random features combination.

Fig. 4.9 provides observed root mean squared error rates as the depth of the tree is observed at 1, 2, 4, 8, 16 and 32 for a Random Forest with 80 trees. Stable value of the error rate is observed at a tree depth of 8 when 32 features are observed. A depth of 16 is required for all other observed feature counts. We observe similar pattern for other tree sizes.

In order to compare the results obtained by applying the Random Forest algorithm to other classification techniques, we perform preliminary classification tests on our dataset with default settings for classifiers in Weka. The number of

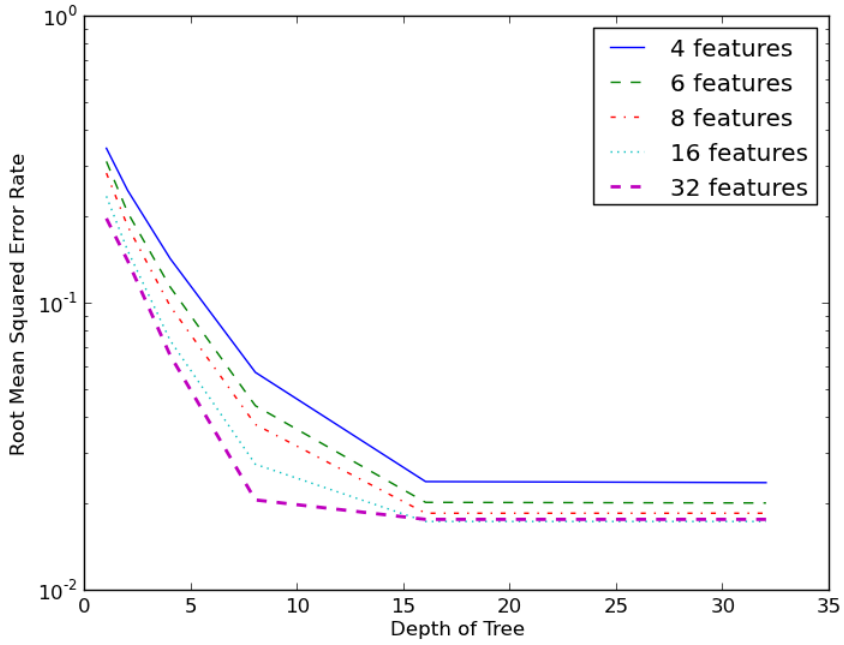

Figure 4.9: Root Mean Squared error rate comparison with 80 trees as depth of tree is varied

misclassified instances of the 48919 points are as shown in Table 4.5. It should be pointed out that all the algorithms listed have associated free parameters which can possibly be tuned to obtain better results. Similarly we have not computed values for other existing classification algorithms such as Support Vector Machines. We consider it outside the scope of this research work as our focus is on testing the parameters of Random Forest for our dataset.

4.4 Discussion

Based on our experimental results on 10 fold cross validation and separate validation set test we make the following observations:

Table 4.5: Misclassification Comparison of Different classifiers

Algorithm	Misclassification
BayesNet	342
NaiveBayes	2361
MultilayerPerceptron	38
J48	47
Decision Stump	2532
Logistic Regression	323
Random Forest	7

- **Random forest sensitive to parameter settings** In our tests, we observe that using the default settings of an algorithm does not yield the best results as is done in all literary work. For 10 fold cross validation a default setting of 10 tree and 6 features compared with maximum depth yields 93 percent true classification. But a 95.74 percent true classification was obtained with 160 trees with 32 features at depth 32.

- **Cross validation results higher than validation set test** Whereas in cross-validation tests, results show true classification at over 90 percent at tree depth greater than 8 as shown in Fig. 4.6, in case of validation set tests, they rarely are higher than 80 percent true classification in general as shown in Fig. 4.5. This difference in result is primarily because there are feature vectors of an application in the validation set, that was not observed in the training set.

- **Class imbalance** Based on comparing classification results provided in Fig. 4.5 that shows performance on a class balanced dataset generated by SMOTE

vs Fig. 4.4 that shows performance with a class imbalance of the original dataset, we observe that the SMOTE generated classifier performed better at tree depth of 4. Thus, using a class balanced dataset would yield results faster. The balanced dataset performed worser than the imbalanced dataset at higher tree depths due to over fitting the model to the training set.

In our experiments, based on a 5-fold cross validation, Random Forest provided an exceptionally high accuracy of over 99.9 percent of the samples correctly classified (see table 4.4). This is is comparison to the 10-fold cross validation tests shown in Fig. 4.6 the highest accuracy obtained was 96.40 percent for trees with 8 features compared at a decision point, at a tree depth of 32. The optimal square-root of the mean-squared-error achieved was 0.0171. The optimal out-of-bag error rate obtained was 0.0002 with a minimum forest size of 40 trees. Based on experimental results on 5-fold cross validation on a previous larger dataset, we can make the following observations for the Android features evaluated:

- **High accuracy of random forest:**

  Random Forest provides an exceptionally high accuracy with over 99.9 percent of the samples correctly classified when trees are allowed to grow to maximum depth. The square-root of the mean-squared-error is 0.0291 or less. The out-of-bag error estimate of Random Forest is acceptably low with 40 trees or higher. It varies between 0.0002 and 0.0004. The best setting based on root MSE value was using 160 trees and 16 random features selected with a Root MSE value of 0.0171. The best setting based on number of incorrect classifications was 160 trees with 8 random features selected with a score of 7 incorrect classifications.

- **More trees are better:** The overall trend shows that the out-of-bag error reduces on average as the number of trees increase. For our data sample, we observe significant better out-of-bag error rates when we have at least 20 trees for the varying random features selected from 4 to 32.

- **Depth of tree required:** Based on experimentation, we observe that for our dataset, we need to construct trees of depth 16. Lower depth than this causes higher number of inaccurate classifications. Measuring trees of depth greater than 16 does not cause a statistically significant change for a given number of trees in the Random Forest algorithm for our dataset. Random Forest provides an accuracy of over 91 percent with a tree depth of 1. With 4 features selected, Random Forest provides greater than 99 percent classification with a depth of 8. With 6 or more features selected, a 99 percent correct classification can be achieved with a depth of 4 for each tree.

- **Lower features per tree better:** For a given forest size, after a certain point, we see that as the number of features are increased for a given tree, the number of incorrect classifications increase. The author in [22] had mentioned that ideally choosing log M features (where M is the size of the total attributes) would yield a good result. In our case, choosing 6 or 8 attributes for a forest gives ideal results for a particular forest size.

- **Misclassified malicious samples:**

  Even though we do not show it in our results table, for all experiments, most of the misclassified cases were due to the malicious class samples misclassified as being of the benign class i.e. there were more false negatives than

false positives. For example the experimental setting yielding the best result with a forest with 160 trees, 8 random features, and depth of 16 for each tree, we observed 5 false negatives and 2 false positives. This is based on observing the generated confusion matrix for each experiment case.

4.5 Summary

The results of these experiments show that the Random Forest algorithm provides true classification results at over 90 percent if we are to observe cross validation results. We also observed that the algorithm is highly sensitive to the parameter values that are chosen. The parameters that we experimented with included the number of decision trees in the Random Forest, the number of random features compared at a decision node for each decision tree, and the depth of each tree in the forest. We observed that choosing the default values of the algorithm, as is done in all research work that have tested with the Random Forest algorithm provide results that are not optimal. We also observed that having good results in cross-validation tests, does not necessarily translate to having good results with a separate validation-set test. Moreover, training a Random Forest classifier on a balanced dataset allows to find better results at lower tree depths. This is beneficial as models of lower depth are smaller in size as shown in Table. 4.2 and take less time to compute.

Chapter 5

Conclusion And Future Work

5.1 Thesis summary

This thesis studies the use of a Java based multi agent system framework based on the JADE platform on Android mobile devices for malware detection. The agent based framework is context aware in order to facilitate better conservation of resources on mobile devices. The framework defines multiple client based agents that reside on the Android device for data gathering and reporting classification reports to the user. Server side agents in the framework are responsible for analysis of features by comparing to pre computed models that have been saved in the system. This client-server approach was taken due to resource limitations on mobile devices in terms of system memory, disk and battery constraints. Moreover, the server side agents can be distributed among multiple servers in the cloud for scalability as they communicate with each other using ACL.

In order to measure the scalability of the multi-agent platform, we have computed the amount of CPU resources used by the agent platform on Android devices.

We also measured the memory requirements to maintain such an active agent system on the Android devices using Android's Dalvik Debug Monitoring Service. Given that many mobile services have restrictions on the amount of data used, we have measured the network resources that are utilized in order to transfer data between the agents on the client and the agents on the server for analysis. All of these measurements are required as there is no clear methodology to measure the resource utilization pattern on mobile devices, since battery usage depends on many external factors, such as external temperature, how long the battery has been used, and the capacity of the battery.

We performed context aware testing by activating Action agents on the Android mobile device based on context. For context based testing we mocked the GPS information used to activate one, or another agent. Similarly we used the network that the Android device was connected to in order to make context aware decisions based on network connection. We launched different agents based on if devices were rooted, and the kind of applications that were installed on the device. For this purpose we installed tools such as RootTools.

In order to detect if Android devices have been infected with malware, the agent framework relies on the use of machine learning classifiers that use pre-built models to make decisions. The pre-built model used for comparison has to be relevant in terms of the features observed. This is because, not all system resources that can be observed on one Android mobile system can be detected on another. This could be caused by the API level of the operating system used by the device, and also the presence of third party analysis tools used to build features.

For this thesis we studied the performance of the Random Forest algorithm on a balanced dataset with a 50-50 split between goodware and malware for the

feature vectors. We started with making a 5-fold cross validation study on our labeled dataset. The study revealed results of over 90 percent true classification rates. Later we revised our experimentation to use 10-fold cross validation on the labeled dataset and also performed a training-validation set study. For this study we performed three runs with random seeding for each combination of the free parameters of the Random Forest algorithm. We observed that whereas cross validation causes results to be over 90 percent true detection, validation set tests rarely cross 80 percent true classification rates. We observed that classification rates vary widely as the free parameters of the algorithm are modified, and that the default setting of algorithm do not yield optimal results.

In our experimentation, we also measured the amount of disk space that will be required to store these pre-built models. We made use of pre-built models for two purposes. One is because once the models are generated, the cost to compare feature vectors is very quick. We measured the cost associated for one feature vector comparison, vs multiple comparisons made from a single call to reach our classification decision. In our experimentation, we could compare against 1350 stored machine learning models in 24.3 seconds for a single feature vector; vs 28.0 seconds for 256 feature vectors. The second reason was that this would allow us to better understand the size of the models generated on Android devices if created with portable versions of Weka, such as the one with Weka for Android devices [53]. The size of our models were between 76 KB and 15.2 MB.

5.2 Future work

The future work related to this thesis can be broken down into two parts. The first is related to the use of machine learning approaches and its potential weaknesses. The

second, is related to the architecture of using multi-agent systems as a detection framework. We now discuss each in the next two sections.

5.2.1 Application of machine learning

As malware in the Android mobile setting is constantly adapting to detection techniques, many of the detection models described here and in the referenced literature have to be tuned to observe new behavior patterns. As mentioned by Allix et al. in [7], [8], and [6], the performance of machine learning classifiers on Android suffer in performance when the test dataset does not exhibit features that the classifiers have not been trained on previously. We now list some future directions that can be taken to refine this work further.

Bayesian optimization for parameter selection

Bayesian optimization [72, 78] is a promising approach for automatically adjusting free parameters in any given algorithm. This optimization methodology through random embedding was developed parallel to when we were developing our approach, and could handle up to a billion dimensions. In the case of Random Forest, the free parameters include: the number of trees in the forest, the number of random features selected at each decision node of a tree, and the maximum depth of each tree. We envision using an objective function that uses the out-of-bag error estimate to guide the Bayesian Optimization algorithm into selecting the right combination of parameters that provide us acceptable settings based on our required threshold. Due to the reason mentioned above, and as the number of free parameters in the Random Forest algorithm are not too many; we did not make use of this approach as we had determined that our accuracies followed a particular pattern and using

this method would not contribute to our results significantly.

Observing more features

The dataset we used primarily focused on the binder API of Android, CPU usage pattern, and memory usage pattern on an Android device emulator. Many papers listed in the related work section used system call tracking as features. Random Forest is relatively immune to increase in the number of features of a feature vector as it requires to observe only log m features of the available m features. Thus, monitoring system call features and more fine grained network level behaviour would allow us to create better detection models. For example, Allix et al. [7] created a set of 2.5 million features constructed from static features gathered from .dex byte code files of Android Executables. Our approach to testing with the parameters of the Random Forest algorithm could be tested on this data. The reason why we have not tested this dataset is because of its size and the limitations of the weka library used for our analysis. The Java Weka library requires that the entire dataset be first copied into memory before a model can be built. Given that the dataset was approximately 250 gigabytes in size, the dataset would require pre-processing to remove unnecessary static features to minimize its size to avoid running out of memory.

Ensemble learning

Ensemble learning creates a learning model by integrating the results of multiple models. Though the Random Forest algorithm is an ensemble learning method that averages the results of multiple models (decision trees), we could create another level of ensemble learners, some of which monitor device at the user level and

others that monitor features at the kernel level. For example, we could create a learning model for *tcpdump* data to monitor network traffic and another to monitor system call data using strace. A foreseeable challenge in this case would be to see how good a classification technique would be in the absence of observed data during testing time. This would be the case if some of the features cannot be observed if a device is unrooted; or a tool for capturing the behaviour is missing in a rooted device. Similar issues will occur if Google decides to remove access to API used for observing behaviour at user level.

Fine grained inspection

The features observed in our dataset are global in nature i.e. all concurrently running applications together impact the features observed. This would cause the data features measured to be very noisy at best without being able to individually break down the impact of each application on the measured features. Similar issues can be identified with [71] as feature vectors were collected by running a single Android application at a time on a real device. One approach we envision would be to normalize the measured feature changes by fine grained inspection of currently running applications. This information was allowed to be queried using the *ActivityManager* class using the *getRunningTasks* method in Android API but was removed in API level 21 (Android Lollipop). This approach would however be challenging as it would require first building the feature vector of each of the individual applications available on the system before being of use.

5.2.2 Use of multi-agent systems

Dynamic loading of agents

Currently, our system requires that all Action agents dynamically launched by the Profile agent reside on the Android device. This requires us to develop a new Android package file every time we create a new agent. The *LoaderBehaviour* class of JADE currently does not support loading Dalvik Bytecode (DEX) bytecode used in Android systems. If DEX bytecode loading is supported by the LoaderBehaviour class, it would allow to load agents not stored on the device.

Using other types of communication framework

There are alternative solutions that can be used instead of agent-based systems. Given that one of the primary reasons for using the agent-based system was to communicate data between clients and servers, this could be handled by using REpresentational State Transfer (REST) web services [64] by passing Extensible Markup Language (XML) or JSON data describing the fields instead of using Agent Communication Language (ACL). In that case, the Android devices would behave as REST clients communicating with the servers for the information. However, given that RESTful services require clients to maintain state instead of servers, the Android clients would have to periodically query the server to see if a classification result was available, causing more network communication and hence battery usage.

Bibliography

[1] C. Aart. *Ontology Bean Generator*, 2015 (accessed 31 Jan, 2015). URL http://protegewiki.stanford.edu/wiki/OntologyBeanGenerator. → pages 35

[2] M. Alam and S. T. Vuong. An intelligent multi-agent based detection framework for classification of android malware. In *Active Media Technology*, pages 226–237. Springer, 2014. → pages iv, v

[3] M. Alam and S. T. Vuong. Performance of malware classifier for android. In *6th IEEE Annual Information Technology, Electronics and Mobile Communication Conference*. IEEE, 2015. → pages iv, v, 4

[4] M. Alam, Z. Cheng, and S. Vuong. Context-aware multi-agent based framework for securing android. In *Multimedia Computing and Systems (ICMCS), 2014 International Conference on*, pages 961–966. IEEE, 2014. → pages iv, v, xii, 42, 43

[5] M. S. Alam and S. T. Vuong. Random forest classification for detecting android malware. In *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing*, pages 663–669. IEEE, 2013. → pages iv, v, xi, 4, 18, 31, 34, 48, 73, 75, 86

[6] K. Allix, T. F. Bissyandé, Q. Jérome, J. Klein, and Y. Le Traon. Empirical assessment of machine learning-based malware detectors for android. *Empirical Software Engineering*, pages 1–29, 2014. → pages 13, 98

[7] K. Allix, T. F. D. A. Bissyande, J. Klein, and Y. Le Traon. Machine learning-based malware detection for android applications: History matters! 2014. → pages 98, 99

[8] K. Allix, Q. Jerome, T. F. Bissyande, J. Klein, R. State, and Y. L. Traon. A forensic analysis of android malware–how is malware written and how it

could be detected? In *Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual*, pages 384–393. IEEE, 2014. → pages 98

[9] B. Amos. *Antimalware*, 2013 (Last Accessed May 15, 2013). URL https://github.com/VT-Magnum-Research/antimalware. → pages 66, 67, 69

[10] B. Amos, H. Turner, and J. White. Applying machine learning classifiers to dynamic android malware detection at scale. In *IWCMC13 Security, Trust and Privacy Symposium*, 2013. → pages 15, 16, 72, 73, 74, 75, 76

[11] E. Andersen. *Busy Box*, 2013 (Last Accessed May 15, 2013). URL https://busybox.net/. → pages 36

[12] Android Su. *Android SuperUser Tools*, 2013 (Last Accessed May 15, 2013). URL http://androidsu.com/superuser/. → pages 36

[13] Android TCPdump. *Android TCP Dump*, 2013 (Last Accessed May 15, 2013). URL http://www.androidtcpdump.com/. → pages 36

[14] Anti Spy Mobile. 2013 (Last Accessed: Aug 14, 2016). URL play.google.com/store/apps/details?id=com.antispycell.connmonitor. → pages 62

[15] Apple. *Apple Store*, 2013 (Last Accessed May 15, 2013). URL http://store.apple.com. → pages 2

[16] M. Asaka, A. Taguchi, and S. Goto. The implementation of ida: An intrusion detection agent system. In *Proceedings of the 11th FIRST Conference*, volume 6. Citeseer, 1999. → pages 11

[17] Z. Aung and W. Zaw. Permission-based android malware detection. *International Journal Of Scientific & Technology Research*, 2(3), 2013. → pages 16

[18] G. Bai, L. Gu, T. Feng, Y. Guo, and X. Chen. Context-aware usage control for android. In *Security and Privacy in Communication Networks*, pages 326–343. Springer, 2010. → pages 19

[19] J. S. Balasubramaniyan, J. O. Garcia-Fernandez, D. Isacoff, E. Spafford, and D. Zamboni. An architecture for intrusion detection using autonomous agents. In *Computer Security Applications Conference, 1998. Proceedings. 14th Annual*, pages 13–24. IEEE, 1998. → pages 10

[20] F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing multi-agent systems with JADE*, volume 7. John Wiley & Sons, 2007. → pages 22

[21] A. Bieszczad, B. Pagurek, and T. White. Mobile agents for network management. *Communications Surveys & Tutorials, IEEE*, 1(1):2–9, 1998. → pages 9

[22] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001. → pages 3, 59, 66, 70, 93

[23] L. Breiman and A. Cutler. *Random Forests*, 2013 (Last Accessed May 15, 2013). URL http://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm. → pages 71

[24] 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. → pages 14, 69

[25] G. Caire, G. Iavarone, M. Izzo, and K. Heffner. *JADE Tutorial: JADE programming for Android*, 2012 (accessed 25 Jan, 2015). URL http://jade.tilab.com/doc/tutorials/JadeAndroid-Programming-Tutorial.pdf. → pages 34

[26] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002. → pages 68, 77

[27] V. Chebyshev and R. Unuchek. *Mobile Malware Evolution 2013*, 2013 (accessed 31 Jan, 2015). URL http://securelist.com/analysis/kaspersky-security-bulletin/58335/mobile-malware-evolution-2013/. → pages 2

[28] Z. Cheng. A multi-agent security system for android platform. Masters Thesis, University of British Columbia, 2012. → pages iv, xii, 33, 36, 42, 43

[29] M. Crosbie and E. H. Spafford. Defending a computer system using autonomous agents. *8th National Information Systems Security Conference*, 1995. → pages 10

[30] N. De Freitas. *Machine Learning*, 2013 (Last Accessed 31 Jan, 2015). URL http://www.cs.ubc.ca/~nando/540-2013/lectures/l1.pdf. → pages 12

[31] J. D. De Queiroz, L. F. R. da Costa Carmo, and L. Pirmez. Micael: An autonomous mobile agent system to protect new generation networked applications. In *Recent Advances in Intrusion Detection*. Citeseer, 1999. → pages 11

[32] A. Desnos. *Androguard: Reverse engineering, Malware and goodware analysis of Android applications*, 2015 (accessed 31 Jan, 2015). URL https://code.google.com/p/androguard/. → pages 8

[33] S. G. Dewan and L. Chen. Mobile payment adoption in the usa: a cross-industry, cross-platform solution. *Journal of Information Privacy & Security*, 1(2):4–28, 2005. → pages 1

[34] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra. Madam: a multi-level anomaly detector for android malware. In *Computer Network Security*, pages 240–253. Springer, 2012. → pages 14, 69

[35] A. Dmitrienko, C. Liebchen, C. Rossow, and A. R. Sadeghi. Security analysis of mobile two-factor authentication schemes. *Intel® Technology Journal*, 18(4), 2014. → pages 3

[36] Elinux. *Procrank Memory Usage Measurement*, 2013 (Last Accessed May 15, 2013). URL http://elinux.org/Android_Memory_Usage#smem_tool. → pages 45

[37] W. Enck, P. Gilbert, B. G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Communications of the ACM*, 57(3):99–106, 2014. → pages 8

[38] S. Erickson. *Root Tools*, 2013 (Last Accessed May 15, 2013). URL https://github.com/Stericson/RootTools. → pages 36

[39] N. Foukia, D. Billard, and P. J. Harms. Computer system immunity using mobile agents. In *HP Openview University Association 8th Annual Workshop*, 2001. → pages 12

[40] N. Foukia, S. Hassas, S. Fenet, and P. Albuquerque. Combining immune systems and social insect metaphors: a paradigm for distributed intrusion detection and response system. In *Mobile Agents for Telecommunication Applications*, pages 251–264. Springer, 2003. → pages 12

[41] W. Glodek and R. Harang. Rapid permissions-based detection and analysis of mobile malware using random decision forests. In *Military Communications Conference, MILCOM 2013-2013 IEEE*, pages 980–985. IEEE, 2013. → pages 13

[42] Google. Google android security overview. Technical report, http://source.android.com/tech/security, May 2013. → pages 69

[43] Google. *Google Play Store*, 2013 (Last Accessed May 15, 2013). URL http://play.google.com. → pages 2

[44] Google. *UI/Application Exerciser Monkey*, 2013 (Last Accessed May 15, 2013). URL http://developer.android.com/tools/help/monkey.html. → pages 66, 72

[45] Google. *Dalvik Debug Monitoring Server*, 2013 (Last Accessed May 15, 2015). URL https://developer.android.com/studio/profile/ddms.html. → pages 41

[46] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1): 10–18, Nov. 2009. ISSN 1931-0145. doi:10.1145/1656274.1656278. URL http://doi.acm.org/10.1145/1656274.1656278. → pages 48, 70

[47] R. Hasan, N. Saxena, T. Haleviz, S. Zawoad, and D. Rinehart. Sensing-enabled channels for hard-to-detect command and control of mobile devices. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 469–480. ACM, 2013. → pages 3

[48] G. G. Helmer, J. S. Wong, V. Honavar, and L. Miller. Intelligent agents for intrusion detection. In *Information Technology Conference, 1998. IEEE*, pages 121–124. IEEE, 1998. → pages 10

[49] D. Kim, J. Kim, and S. Kim. A malicious application detection framework using automatic feature extraction tool on android market. In *3rd International Conference on Computer Science and Information Technology (ICCSIT'2013)*, pages 4–5, 2013. → pages 13, 69

[50] C. Krügel and T. Toth. *Flexible, mobile agent based intrusion detection for dynamic networks*. European Wireless, 2002. → pages 11

[51] C. Krügel, T. Toth, and E. Kirda. Sparta. In *Advances in Network and Distributed Systems Security*, pages 187–198. Springer US, 2002. → pages 11

[52] C. Li, Q. Song, and C. Zhang. Ma-ids architecture for distributed intrusion detection using mobile agents. In *Proceedings of the 2nd International Conference on Information Technology for Application (ICITA 2004)*, pages 451–455, 2004. → pages 11

[53] P. Liu, Y. Chen, W. Tang, and Q. Yue. *Mobile WEKA as Data Mining Tool on Android*, pages 75–80. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-27951-5. doi:10.1007/978-3-642-27951-5_11. URL http://dx.doi.org/10.1007/978-3-642-27951-5_11. → pages 97

[54] F. Maggi, A. Valdi, and S. Zanero. Andrototal: A flexible, scalable toolbox and service for testing mobile malware detectors. In *Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices*, pages 49–54. ACM, 2013. → pages 17

[55] P. Mell and M. McLarnon. Mobile agent attack resistant distributed hierarchical intrusion detection systems. Technical report, DTIC Document, 1999. → pages 11

[56] M.Nikraz, G. Caire, and P.A.Bahri. A methodology for the development of multiagent systems using the jade platform. In *International Journal of Computer Systems Science and Engineering*, volume 21, pages 99–116, 2006. → pages 25

[57] A. Moreno, A. Valls, and A. Viejo. *Using JADE-LEAP implement agents in mobile devices*. Universitat Rovira i Virgili. Departament d'Enginyeria Informàtica, 2003. → pages 19

[58] K. P. Murphy. *Machine learning: a probabilistic perspective*. 2012. → pages 12

[59] M. Musen. *The Protege project: A look back and a look forward. AI Matters.*, volume 4. Association of Computing Machinery Specific Interest Group in Artificial Intelligence, June 2015. doi:10.1145/2557001.25757003. → pages 35

[60] 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. → pages 8

[61] PayByPhone. 2013 (Last Accessed May 15, 2013). URL http://www.paybyphone.com/. → pages 1

[62] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2013*, 2014. URL http://www.internetsociety.org/doc/execute-analyzing-unsafe-and-malicious-dynamic-code-loading-android-applications. → pages 2

[63] G. Ramachandran and D. Hart. A p2p intrusion detection system based on mobile agents. In *Proceedings of the 42nd annual Southeast regional conference*, pages 185–190. ACM, 2004. → pages 11

[64] L. Richardson and S. Ruby. *RESTful web services*. " O'Reilly Media, Inc.", 2008. → pages 101

[65] D. Rosenberg. *It's Bugs All the Way Down*, 12 2011 (Last Accessed: Sep 11, 2015). URL http://vulnfactory.org/blog/2011/12/05/carrieriq-the-real-story/. → pages 3

[66] A. Santi, M. Guidi, and A. Ricci. Jaca-android: an agent-based platform for building smart mobile applications. In *Languages, Methodologies, and Development Tools for Multi-Agent Systems*, pages 95–114. Springer, 2011. → pages 10

[67] A. D. Schmidt, H. G. Schmidt, J. Clausen, K. A. Yuksel, O. Kiraz, A. Camtepe, and S. Albayrak. Enhancing security of linux-based android devices. In *in Proceedings of 15th International Linux Kongress. Lehmann*, 2008. → pages 8

[68] D. Schreckling, J. Köstler, and M. Schaff. Kynoid: real-time enforcement of fine-grained, user-defined, and data-centric security policies for android. *Information Security Technical Report*, 17(3):71–80, 2013. → pages 8

[69] A. Shabtai, Y. Fledel, and Y. Elovici. Securing android-powered mobile devices using selinux. *IEEE Security & Privacy*, 8(3):36–44, 2010. → pages 8

[70] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google android: A comprehensive security assessment. *IEEE Security and Privacy*, 8(2):35–44, 2010. → pages 8

[71] 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. → pages 15, 69, 100

[72] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012. → pages 98

[73] S. J. Stolfo, A. L. Prodromidis, S. Tselepis, W. Lee, D. W. Fan, and P. K. Chan. Jam: Java agents for meta-learning over distributed databases. In *KDD*, volume 97, pages 74–81, 1997. → pages 11

[74] Telecom Italia Lab. *Jave Agent Development Framework*, 2013 (Last Accessed May 15, 2013). URL http://jade.tilab.com. → pages 18

[75] Trend Micro. *Android Malware Believe the hype*, 2013 (Last Accessed May 15, 2013). URL http://countermeasures.trendmicro.eu/android-malware-believe-the-hype. → pages 2

[76] TrustGo Security. *New Virus SMSZombie.A Discovered by TrustGO Security Labs*, 2012 (Last Accessed: Sep 15, 2015). URL http://blog.trustgo.com/SMSZombie/. → pages 2

[77] S. T. Vuong and M. S. Alam. Advanced methods for botnet intrusion detection systems. In *Intrusion Detection Systems*. InTech, 2011. doi:10.5772/15401. URL http://www.intechopen.com/books/intrusion-detection-systems/advanced-methods-for-botnet-intrusion-detection-systems. → pages iv, vi

[78] Z. Wang, M. Zoghi, F. Hutter, D. Matheson, and N. De Freitas. Bayesian optimization in high dimensions via random embeddings. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 1778–1784. AAAI Press, 2013. → pages 98

[79] C. Xiang, F. Binxing, Y. Lihua, L. Xiaoyi, and Z. Tianning. Andbot: towards advanced mobile botnets. In *Proceedings of the 4th USENIX conference on Large-scale exploits and emergent threats*, pages 11–11. USENIX Association, 2011. → pages 3

[80] Y. Zhauniarovich, G. Russello, M. Conti, B. Crispo, and E. Fernandes. Moses: supporting and enforcing security profiles on smartphones. *IEEE Transactions on Dependable and Secure Computing*, 11(3):211–223, 2014. → pages 19

[81] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci. StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY '15, 2015. to appear. → pages 13

[82] 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. → pages 3, 13