Android Malware Detection through Machine Learning on Kernel Task Structure

by

Xinning Wang

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
December 16, 2017

Keywords: Android Phones, Malware Detection, Machine Learning, In-memory classification, RBF network, EBP network

Copyright 2017 by Xinning Wang

Approved by

Bo Liu, Chair, Assistant Professor of Computer Science and Software Engineering
Kai Chang, Professor of Computer Science and Software Engineering
Wei-Shinn Ku, Associate Professor of Computer Science and Software Engineering
Sanjeev Baskiyar, Associate Professor of Computer Science and Software Engineering

Abstract

The popularity of free Android applications has risen rapidly along with the advent of smart phones. This has led to malicious Android apps being involuntarily installed, which violate the user privacy or conduct attack. According to the survey of Android malware from Kaspersky Lab, the proportion of malicious attacks for Android software has increased by a factor of two since 2009. Therefore malware detection on Android platforms is a growing concern because of the undesirable similarity between malicious behavior and benign behavior, which can lead to slow detection, and allow compromises to persist for comparatively long periods of time in infected phones. Meanwhile a huge number of malware detection techniques have been proposed to address the serious issue and safeguard Android systems. In order to distinguish malicious apps from Android software, the traits of malware applications must be tracked by software agents or built-in programs. However, these researchers only utilize a short list of the Android process features without considering the completeness and consistence of the entire system level information.

In this dissertation, we present a multiple dimensional, kernel feature-based framework and feature weight-based detection (WBD) designed to categorize and comprehend the characteristics of Android malware and benign apps. Furthermore, our software agent is orchestrated and implemented for data collection and storage to scan thousands of benign and malicious apps automatically. We examine 112 kernel attributes of executing the task data structure in the Android system and evaluate the detection accuracy with a number of datasets of various dimensions. We observe that memory- and signal-related features contribute to more precise classification than schedule-related and other descriptors of task states listed in this dissertation. Particularly, memory-related features provide fine-grain classification policies for preserving higher classification precision than the signal-related

features and others. Furthermore, we study and evaluate 80 newly infected attributes of the Android kernel task structure, prioritizing the 70 features of most significance based on dimensional reduction to optimize the efficiency of high-dimensional classification. Our experiments demonstrate that, as compared to existing techniques with a short list of task structure features, our method can achieve 94%-98% accuracy and 2%-7% false positive rate, while detecting malware apps with reduced-dimensional features that adequately abbreviate online malware detections and advance offline malware inspections.

To strengthen the online framework on a parallel computing platform, we propose a Spark-based Android malware detection framework to precisely predict the malicious applications in parallel. Apache Spark, as a popular open-source platform for large-scale data, has been used to deal with iterative machine learning jobs because of its efficient parallel computation and in-memory abstraction. Moreover, malware detection on Android platforms requires to be implemented in a data-parallel computation platform in consideration of the rapid increase of data size of collected samples. We also scrutinize 112 kernel attributes of kernel structure (*task_struct*) in the Android system and evaluate the detection precision for the whole datasets with different numbers of computing nodes on Apache Spark platform. Our experiments demonstrate that, our technique can achieve 95%-99% of the precision rate with a faster computing speed by a Decision Tree Classifier on average, the other three classifiers lead to a lower precision rate while detecting malware apps with the in-memory parallel-data.

We have designed a Radial Basis Function (RBF) network-based malware detection technique for Android phones to improve the accuracy rate of classification and the training speed. The traditional neural network with the Error Back Propagation method cannot recognize the malicious intrusion through Android kernel feature selection. The RBF hidden centers can be dynamically selected by a heuristic approach and the large-scale datasets of 2550 Android apps are gathered by our automatic data sample collector. We implement the

algorithms of the RBF network and the Error Back Propagation (EBP) network. Furthermore, compared to the traditional neural network, the EBP network which achieves 84% accuracy rate, the RBF network can achieve 94% accuracy rate with the half of training and evaluation time. Our experiments demonstrate the RBF network can be used as a better technique of the Android malware detection.

Acknowledgments

First, I want sincerely to thank my advisor Dr. Liu for his thorough academic guidance, patient cultivation, encouragement and continuous support during my doctoral study. I have been really fortunate to become his student and conduct interesting and cutting-edge research work in the outstanding academic environment he created in the research lab. As my advisor, he has been helping me identify novel research topics and solve critical challenges, and giving me all kinds of precious opportunities to hone my skills, broaden my horizons and shape my professional career. He has also been a most helpful friend of me, helping me in my life and encouraging me during the tough moments. I greatly appreciate his priceless time and efforts for nurturing me during my Ph.D experience.

Second, I would also like to thank my committee members: Dr. Chang, Dr. Ku and and Dr. Baskiyar, my university reader Dr. Fan and Dr. Skjellum. Their precious suggestions and patient guidance help to improve my dissertation. Third, I feel really grateful to my group-mates: Austin Hancock, Ye Wang, Tian Liu. Their cooperation in work and help in life make Auburn cyber security lab a big and warm family and an excellent place where we learn, create, improve and enjoy.

My deepest gratitude and appreciation go to my husband, my parents, my parents-in-law, my brother, and my son. They are the charming gardeners who help me grow strong and make my life blossom. Their love and sacrifice have paved this long journey for me to pursue my dreams.

Table of Contents

Abstract ii

Acknowledgments v

List of Figures ix

List of Tables xiii

1 Introduction 1

1.1 Background, Opportunity,and Challenges 1

1.2 Dissertation Statement 3

1.3 Assumptions and Definition of Terminology 4

1.3.1 Measurement of Binary Classification Effectiveness 4

1.3.2 Receiver Operating Characteristic 5

1.4 Overview of Approaches to the Solution 5

1.5 Contribution 7

1.6 Impacts 8

1.7 Structure of this Dissertation 9

2 Literature Review 11

2.1 Android Malware 11

2.2 Apache Spark architecture 13

2.3 Artificial Neural Network 15

2.4 Android PCB kernel structure 16

2.5 112 Android Kernel Variables 18

2.6 Brief Summary of Previous Malware Detection through Behavior 21

3 Problem Statement 23

3.1 Android Architecture 23

3.2 Dynamic Android Malware Detection in Linux Kernel Layer 24
3.3 Static Android Malware Detection in Other Layers 25
3.4 Challenges in TstructDroid and Our Goals 27
3.5 In-Memory Large-Scale Data Training 28
3.6 Artificial Neural Network 29
4 System Design 30
4.1 Overview of Multiple Kernel Features 30
4.1.1 Overview of Malware Behavior in Kernel Level 30
4.1.2 A Case Study of Features in Malware Vs. Goodware Distribution . . 31
4.1.3 Measurements of Multiple Dimensional Kernel Features 35
4.2 Data Processing 43
4.2.1 Data Cleaning and Data Filling 43
4.2.2 Dimensional Reduction Methods 44
4.3 Local Machine Learning Methods 49
4.4 Parallel Malware Detection 50
4.4.1 In-Memory Classification 50
4.4.2 Parallel Classifiers 51
4.5 Designs of Neural Network 53
4.5.1 Traditional Neural Network (EBP) 53
4.5.2 Enhanced Neural Network 55
4.5.3 RBF Network Design and Implementation 57
4.6 Multiple Dimensional Kernel Feature Collector 60
4.7 Normalized Feature Weights 62
4.7.1 Distribution of Normalized Feature Weights 62
4.7.2 Details of Normalized Feature Weights 66
5 Analysis of Experimental Results 70
5.1 Experimental Configuration 70

5.1.1 Experimental Setup for Data Collection 70
5.1.2 Experimental Setup for Data Processing 71
5.1.3 Experimental Setup for Classification 71
5.2 Results of Local Classifiers 73
5.2.1 Distribution and Analysis of Kernel Features 73
5.2.2 Comparison of newly infected and previously infected parameters . . 75
5.2.3 Cross-Validation Results 78
5.3 Results of Parallel Classifiers 88
5.3.1 Execution Time 88
5.3.2 Classification Precision 91
5.4 Evaluation of RBF 95
5.4.1 Resource Allocation Results 95
5.4.2 Comparison of Accuracy Rate 96
6 Summary and Future Work 98

List of Figures

2.1 Framework of Spark on Mesos 13

2.2 Apache Spark Ecosystem 15

3.1 Android Architecture 23

4.1 2D Distribution of 100 benign and 100 malware samples with Shared_vm and Total_vm 32

4.2 2D Distribution of 100 benign and 100 malware samples with Signal_nvcsw and Total_vm 33

4.3 2D Distribution of 100 benign and 100 malware samples with Signal_nvcsw and Shared_vm 34

4.4 3D Distribution of 100 benign and 100 malware samples: with the increase of the number of dimensions, benign and malware samples cluster together in different areas. 35

4.5 total_vm 36

4.6 exec_vm 37

4.7 reserved_vm 37

4.8 shared_vm 38

4.9 map_count 38

4.10 hitwater_rss 39

4.11 nivcsw 39

4.12 nvcsw 40

4.13 maj_flt 40

4.14 nr_ptes 41

4.15 signal_nvcsw 41

4.16 stime 42

4.17 ROC Curve of four classifiers 52

4.18 Architecture of Error Back Propagation Algorithm 53

4.19 Architecture of Radial Basis Function 54

4.20 Overview of Multiple Dimensional Kernel Feature's (Raw Data) Collector. In (b), Message Communication Module in Local Computer . In (c), Data Processing Module in Android Kernel. 61

4.21 Normalized Weights Distribution of 112 Parameters with PCA method (**mem_info** & **signal_info** top 2 most popular) 63

4.22 Normalized Weights Distribution of 112 Parameters with Correlation method (**mem_info** & **signal_info** top 2 most popular) 63

4.23 Normalized Weights Distribution of 112 Parameters with Chi-square method (**mem_info** & **signal_info** top 2 most popular) 64

4.24 Normalized Weights Distribution of 112 Parameters with Info Gain method (**mem_info** & **signal_info** top 2 most popular) 64

5.1 Comparison of Currently Infected Parameters and Previously Infected Parameters 75

5.2 Non-Zero Normalized Weights of Previously-Infected Task Parameters (There are 32 previously-infected task parameters shown in Fig. 5.1 in detail.) 76

5.3 Non-Zero Normalized Weights of Newly-Infected Task Parameters (There are 80 newly-infected/currently infected task parameters shown in Fig. 5.1 in detail.) 76

5.4 True Negative Rate by Decision Tree With the Increasing Number of Selected Features: VBD is proposed in [75] and WBD denotes our methods, on average WBD achieves 6% improvement of TN. 78

5.5 True Positive Rate by Decision Tree With the Increasing Number of Selected Features: VBD is proposed in [75] and WBD denotes our methods, on average WBD achieves 12% improvement of TP. 78

5.6 Accuracy Rate by Decision Tree With the Increasing Number of Selected Features: VBD is proposed in [75] and WBD denotes our methods, on average WBD achieves 10% improvement of accuracy. 79

5.7 True Negative Rate by Naive Bayes Kernel With the Increasing Number of Selected Features: Correlation method leads to the highest TN than PCA, Chi-square, and Info Gain on average. 80

5.8 True Positive Rate by Naive Bayes Kernel With the Increasing Number of Selected Features: PCA achieves the best TP compared to others on average. . . . 80

5.9 Accuracy Rate by Naive Bayes Kernel With the Increasing Number of Selected Features: 4 methods achieves the similar accuracy results on average, PCA achieves slightly higher accuracy. 81

5.10 True Negative Rate by Decision Tree With the Increasing Number of Selected Features: Correlation and Chi-square methods lead to the highest TN than PCA and Info Gain. 82

5.11 True Positive Rate by Decision Tree With the Increasing Number of Selected Features: Chi-square method achieves the best TP compared to others on average. 82

5.12 Accuracy Rate by Decision Tree With the Increasing Number of Selected Features: 4 methods achieve the similar accuracy results on average, Chi-square can achieve a bit higher accuracy. 83

5.13 True Negative Rate by Neural Net With the Increasing Number of Selected Features: Info Gain method leads to the highest TN than PCA, Correlation, and Chi-square. 84

5.14 True Positive Rate by Neural Net With the Increasing Number of Selected Features: Correlation method achieves the best TP compared to others on average. 84

5.15 Accuracy Rate by Neural Net With the Increasing Number of Selected Features: 4 methods achieves the similar accuracy results on average, Correlation can achieve slightly higher accuracy. 85

5.16 Execution Time (min) of DT Classifier 88

5.17 Execution Time (min) of LR Classifier 89

5.18 Execution Time (min) of SVM Classifier 89

5.19 Execution Time (s) of NB classifier 90

5.20 Classification Precision by DT Classifier 91

5.21 Classification Precision by LR Classifier 92

5.22 Classification Precision by SVM Classifier 93

5.23 Classification Precision by NB Classifier 93
5.24 Precision Comparison of DT, NB, LR, and SVM Classifiers 94
5.25 Memory Usage of RBF and EBP 95
5.26 CPU Usage of RBF and EBP 95
5.27 Accuracy Rate of RBF and EBP with Hidden Neurons 97

List of Tables

2.1 112 Android Kernel Features 18

4.1 Key Variables of Active Processes 36

4.2 Contingency Table of i-th Feature and Category in Training Set **X** 49

4.3 Normalized Weights of 112 Task Parameters with PCA, Correlation, Chi-square and Info Gain 66

5.1 Hadoop Configurations 72

5.2 Spark Configurations 72

5.3 Distribution of 112 Task Parameters Normalized Weights with PCA, Correlation, Chi-square and Info Gain Methods: **mem_info**, the most correlated feature set for classification, achieves the maximum number of large weights between 50% and 100% in 4 different techniques, next is **signal_info**, **sche_info**, **others** and **task_state** also contribute to precise classification. The details are located in Table 5.4. 74

5.4 TP Rate, TN Rate and Accuracy Rate According to Select Different Numbers of Features by PCA, Correlation, Chi-square and Info Gain with 3 Different Machine Learning Algorithms (Decision Tree, Naive Bayes and Neural Network) 86

Chapter 1

Introduction

1.1 Background, Opportunity,and Challenges

The growing market share of Android smart phones has been accompanied by the unprecedented rise of malicious threats, including web-based threats and application-based threats [2]. As compared to web-based malicious threats, which exploit vulnerable websites to inject malware into users' phones, application-based threats focus on masquerading as legitimate apps in order to deceive users into installing and executing them. According to a 2015 survey of Android security [82], there were numerous shortcomings in its system security, in part because of its open-source framework, install-time permission, and because of the lack of isolation with third-part applications. As a result, a large number of Android devices have become routinely susceptible to malware.

In terms of severe damage inflicted by malicious apps, attackers regularly attempt to steal user private information, obtain administrator privilege, or misuse resources. Recently, Kaspersky Lab reported that the proportion of malicious attacks in 2015 for Android software increased by a factor of two in Trojan Banking malware families [42]. Consequently, a myriad of malware detection techniques [26, 49, 95, 55, 31] have been proposed to address this issue and safeguard Android systems. Among them, kernel-based detection [49] has grown in popularity because this approach can audit all applications of an Android phone and obtain detailed log information from a Linux[1] kernel layer. Moreover, kernel feature-based malware prediction achieves a detecting accuracy rate of 95%, analyzing task structures [76] in the Linux system rather than the Android system.

[1] Linux 14.04.1-Ubuntu

Normally, the Android permission system denies access to user sensitive data (SMS, business (trade secrets, contracts, or call information), etc.) from potentially malicious apps. Through a straightforward SMS operation in Google Play, a SMS related permission can not grant the access of sending messages or receiving messages to an untrustful app from threatening apps websites. In addition, when installing an app that attempts to be granted permission of important business data, users can adjust and limit the app permissions of disclosing the business information. However, some malicious applications, authorized unwittingly by users, such as Trojan horse apps which masquerade as legitimate apps, are difficult to detect only via def-use based behavior analysis [53].

On the other hand, because of middleware code's obfuscation and polymorphism, signature- or configuration-based malware detection [79] also face constraints of application communication processing based on Binder IPC or shared memory mapping. Therefore, kernel feature-based malware detection technique [76] is considered as an effective option of identifying robust features of a running process. This technique is classified into two categories: static analysis without executing programs and dynamic analysis of executing programs [38], both of which vary in terms of performance.

Under kernel feature-based malware detection, the number of features (attributes of task structure in the kernel layer) influences the correctness and scalability of malware detection. In [76], a short list of kernel variables (16 attributes) is used to identify malicious applications. However, such few attributes **may** cause overfitting issue [37] as the size of data rises. Moreover, cumulative variance of each feature after discrete transformation heavily degrades the performance of malware detection from both theory and experimental perspectives [75]. We observe that a small number of kernel features dataset may lead to a low accuracy rate of Android malware detection, and incur the limitation of the feature extract and feature select if acquiring less kernel features. Thus, there is still a need of malware detection with high dimensional features that can lead to a good overview for all relevant features of the

current task structure, and more importantly, sustain stable results of malware detection in case of training model overfitting.

1.2 Dissertation Statement

In this study, we investigate and explore a multiple dimensional kernel feature-based solution for malware detection in an Android platform. Additionally, we examine the genetic footprints of 112 kernel features (**task_struct**) of Android smart phones and empirically analyze the influences of memory- and signal- related features. Furthermore, we calculate the weights of 112 features for dimension reduction with linear and nonlinear algorithms [40] and compare their results to provide an insight to predict impacts of newly-injected attributes of the task structure. Our experimental results demonstrate that the multiple dimensional kernel-based malware detection can reduce the false positive rate, while choosing the right number of features and applying proper algorithms. Our methods can be used to detect the Android malware locally with the 112 kernel features or reduced attributes.

Furthermore, in order to retain the scalability of the large-scale data computation, we propose a parallel malware detection framework to analyze and evaluate Android datasets. Our methods can systematically examine the 112 kernel features on physical Android phones and categorize these kernel features. Moreover, from our experiments, the sensitivities of algorithms (Receiver Operating Characteristic (ROC) space), illustrate which algorithm can achieve the best classification precision. Our parallel methods can efficiently find the best algorithm to detect the Android malware online by transmitting the data to remote server.

In order to satisfy the expected requirements of Neural Network in the exascale computation, our RBF network-based Android malware detection method, in which the heuristic approach of clustering, K-means algorithm, is used to select the initial clustering centers. Our methods use the Euclidean distances among the large amount of data points to measure the similarity of malicious or benign samples. Our methods can capture more characteristics

from the undisciplined data samples and lead to a good classification result. Additionally, our methods can be used to improve the classification performance of the traditional neural network with the large-scale dataset.

1.3 Assumptions and Definition of Terminology

1.3.1 Measurement of Binary Classification Effectiveness

In binary classification, classifying an app as benign is commonly accounted to be positive and vice versa. Likewise, a malware app is to be negative. The performance of the binary classification is measured and quantified by the four elements in a confusion matrix: True Positive (TP) (the proportion of identifying benign instances as being nonmalware), True Negative (TN) (the proportion of recognizing malicious instances as being malware) , False Positive (FP) (the proportion of identifying malicious instances as being benign), and False Negative (FN) (the proportion of identifying benign instances as being malware). Generally, a good machine learning algorithm should prevent FP and FN, and conserve the TP and TP.

High TP rate and TN rate, in malware detection of Android devices, indicate that malicious and benign instances are mostly categorized as correct categories. In order to reduce the risk of posing a threat, a high FN rate and low FP rate are useful for ruling out malicious apps. Indeterminate instances are treated as FN cases without any damage to the whole system, unlike FP case resulting in serious damage to customer or system. The standard metrics used in our experiments are shown as below:

True Positive (TP) Rate This measures the proportion of the benign that is recognized as nonmalware, as calculated by the equation: $TP_rate = TP/(TP + FN)$.

True Negative(TN) Rate This represents the proportion of the malicious that is classified as malware using the equation: $TN_rate = TN/(TN + FP)$.

Accuracy Rate (Acc.) This evaluates the portion of all the benign and malware which are correctly categorized as calculated with the equation: $Acc._rate = (TP + TN)/(TP + TN + FP + FN)$.

In addition, we can use the above equations to derive the FP rate and FN rate as shown as the following: $FP_rate = 1 - TN_rate$ and $FN_rate = 1 - TP_rate$.

1.3.2 Receiver Operating Characteristic

A Receiver Operating Characteristic (ROC) visually illustrates the performance of a binary classifier in graphical plot. Commonly, the curve is plotted by the true positive (TP) rate of the vertical axis against the false positive (FP) rate of the horizontal axis with different discrimination threshold. For each sample, there is a probability value which decides whether this sample is positive or not. If the probability of the testing sample is more than or equal to the discrimination threshold ([0, 1]), this testing sample is recognized the positive one. Moreover, all the testing samples satisfying this condition are recognized the positive ones regardless of their original categories. The others are recognized the negative ones. While choosing different discrimination thresholds, we can achieve different groups of FP rate and TP rate. Through a ROC analysis of several models, the optimal model can be selected possibly by comparing the area under the ROC curve, which is a natural method to analyze the models. Area Under the ROC Curve (AUC) can compare the classifier performance with a scalar value or a graphics demonstration. In general, the larger AUC value represents the better and more accurate results from a classifier.

1.4 Overview of Approaches to the Solution

Firstly, we explore a multiple dimensional kernel feature-based solution for malware detection in an Android platform and examine the genetic footprints of 112 kernel features (**task_struct**) of Android smart phones and empirically analyze the influences of memory-

and signal- related features. Furthermore, we calculate the weights of 112 features for dimension reduction with linear and nonlinear algorithms [40] and compare their results to provide an insight to predict impacts of newly-injected attributes of the task structure. Our experimental results demonstrate that the multiple dimensional kernel-based malware detection can reduce the false positive rate, while choosing the right number of features and applying proper algorithms.

Apache Spark [93], as a popular large-scale data processing framework, has been used to improve the performance of iterative machine learning algorithms in parallel. Due to a read-only collection of objects, Resilient Distributed Dataset (RDD), Spark can easily cache parallel data in memory and iteratively exploit RDD in parallel operations, which eliminates the overhead of I/O communications. Therefore, we propose a Spark-based malware detection framework to effectively distinguish malware in parallel. We systematically examine 112 kernel features on physical Android phones and categorize these kernel features. Furthermore, we evaluate the methods of linear and nonlinear machine learning algorithms, including Naive Bayes (NB), Decision Tree (DT), Support Vector Machine (SVM) and Logistic Regression (LR) to identify malicious apps.

We propose a RBF network-based Android malware detection method for the large-scale dataset of Android apps, in which the heuristic approach of clustering, K-means algorithm, is used to select the initial clustering centers. The Euclidean distances among the large amount of data points measure the similarity of malicious or benign samples. In Artificial Neural Network (ANN) approaches, the Android malware can be detected based on techniques of machine learning through training linear or nonlinear classification models. With the advantage of ANN detection approaches, ANN successfully recognizes the malicious intrusion through feature selection and analysis of critical infrastructures [57, 32, 81].

In both of our solutions, we use the 112 kernel features of **task_struct** to construct the training models. However, for the multiple dimensional kernel-based method, we only use 550 Android applications (275 malicious apps and 275 benign apps) to train the models due

to the constrains of computation resources. Both spark-based malware detection and the RBF network train the classification models with 2550 Android applications (1275 malicious apps and 1275 benign apps).

1.5 Contribution

At first, we only collect a small number of Android dataset, 550 Android apps (15,000 $\times$ 550 records). The computation needs few CPU cores (4 cores in our experiment) and a small memory (16 GB). We implement the design and analyze the results in the powerful computers. Therefore, We propose a multiple dimensional kernel feature-based framework to detect unknown malware apps dynamically in Android platforms. We systematically examine as many as 112 kernel features from 275 malware apps and 275 nonmalware apps on physical phones facilitated by our automated software agent of collecting their information. Furthermore, we conduct a comprehensive analysis of these kernel features and compare 112 task attributes (parameters) with 32 previously infected attributes and analyze their normalized weights' distribution to discover 112 task attributes' impacts on the malware detection.

Because we have collected a large number of Android dataset in total 2550 android apps (15,000 $\times$ 2550 records), a more powerful platform is required to effectively calculate the probabilities of data samples. We implement and deploy the previous framework to the parallel platform, Apache Spark. Our further studies are summarized as following We propose a Spark-based malware detection framework to effectively identify malware. We systematically examine the 112 kernel features from 1275 malware apps and 1275 benign apps on physical Android phones and categorize these kernel features. Moreover, our experiments show the sensitivities of algorithms change rapidly from Receiver Operating Characteristic (ROC) space. We evaluate the methods of linear and nonlinear machine learning algorithms, including Naive Bayes (NB), Decision Tree (DT), Support Vector Machine (SVM) and Logistic Regression (LR) to identify malicious apps.

In addition, we find the traditional artificial neural network with Error Back Propagation (EBP) technique detect the malware with a lower accuracy rate when the number of Android dataset increases greatly. Thereby, we propose a RBF (Radial Basis Function) network-based Android malware detection method, in which heuristic approach of clustering, K-means algorithm, is used to select the initial clustering centers. The Euclidean distances among the large amount of data points measure the similarity of malicious or benign samples. We implement and evaluate the methods of RBF network and EBP network. Our experiments demonstrate, compared to the EBP network, the RBF network can achieve an higher accuracy rate and reserve less resource allocation and execution time.

1.6 Impacts

At first, we analyze the performance issues for selecting relevant features that are effective for detecting malicious apps on the Android platform. Accordingly, we design a multiple dimensional kernel feature-based malware detection infrastructure and implemented a multiple dimensional kernel feature's collection agent so as to dynamically collect, transfer, and store our 112-dimension data. We have examined 275 malware apps each of which has 15,000 instances and 275 benign apps with the same number of instances. The effective dimensional reduction algorithms, PCA, Correlation, Chi-square and Info Gain, are also employed to dig out the more important features to malware detection. The results show that, by using more signal- and memory-related features of Android kernel, classifiers of Naive Bayes, Decision Tree and Neural Network efficiently achieve the 94%-98% of accuracy rate and less than 10% of false positive rate. In contrast to Naive Bayes , Decision Tree and Neural Network can predict more precisely the malicious apps while avoiding the issue of overfitting. These results demonstrate that characterization of kernel features is directly relevant to predicting the malware presence accurately.

Secondly, we propose a Spark-based malware detection framework. The Spark-based malware detection architecture accurately deals with the original data sample from the data

collector and efficiently predict the malicious behaviors in memory. To the end, this work demonstrates the sensitiveness of NB, DT, SVM and LR classifiers on Apache Spark platform, in which the DT classifier can preserve a higher precision rate and eliminate the execution cost. Moreover, our Spark-based malware detection technique improves the performance when the data size dramatically increases and decreases the time consumption caused by frequent I/O communications. In summary, our results indicate the parallel DT classifier is the best algorithm to detect Android malware with the most accurate precision and the lowest cost.

Finally, we proposed a RBF network based malware detection technique with a heuristic approach of clustering. To measure the similarity in Android datasets, the K-means algorithm calculates the center's position for initializing the hidden neurons of the RBF network, which assigns each data point from the large-scale dataset into different regions. According to the initialized hidden centers, the RBF network can quickly and precisely compute the positions for unknown data samples through the correct Gaussian functions. Our results demonstrates that the RBF network can preserve a higher accuracy rate with less execution cost and time. Moreover, compared to the EBP network, the RBF network improves the performance for the exascale computation of the large-scale dataset. Therefore the RBF network is proved to improve the classification performance while the traditional neural network can not meet the criteria of availability or performance for exascale data computation.

1.7 Structure of this Dissertation

The rest of this dissertation is organized as follows. Chapter 2 introduces related research of malware detection, including malware software, benign software, Apache Spark architecture, Artificial Neural Network, and Process Control Block (PCB) Task Kernel Structure. Chapter 3 illustrates the problems of Android malware detection. Chapter 4 presents the designs of the local malware detection, parallel malware detection and the RBF network-based

malware detection. Chapter 5 shows the analysis of the experimental results and Chapter 6 shows the summary of this study and some suggestions of future work.

Chapter 2

Literature Review

This chapter introduces several areas of research which are closely related to our designs, including Android malware, Apache Spark architecture, Artificial Neural Network, and Android PCB kernel structure.

2.1 Android Malware

The penetration of malware applications [62] in Android Phones is categorized into three types: repackaging, downloading, and updating. Among these, repackaging legitimate apps and hiding malicious code in them is the most common method to fool the user to install malware apps. Customers accidentally update nonmalware apps, injecting pieces of malicious code and download malicious apps that camouflage themselves as benign (*e.g.,* Trojan horses).

To "trojanize" a well-known legitimate app, the attacker may employ highly effective renaming, utilize executable wrappers, manipulate source code, and even disguise items through polymorphic code [78]. For example, an attacker may change the name of malicious code to the similar name into the system process so that the malicious appears to belong to the normal process. More seriously, the attacker can open a backdoor to the system allowing for remote administrations. Trojans as the malicious programs, unlike computer virus and worms injecting themselves into other programs, disguise themselves to be unsuspicious and mislead the users of the true intent. Here we provide a brief description of Trojans to gain the obvious characteristics of malicious programs.

Trojan.DDoS

The DDos (Denial of Service) Trojan can form a botnet from injected computers [6]. The program can attack SSH connection, Linux executable files and encryption methods. Initially, the DDos Trojan attempts to destroy SSH credentials of the root account. If successfully breach the root credentials, then the Trojan program is installed through a shell script.

Trojan.Ransom.Gen

The Trojan can disguise a common email or social connection to attack computer users and steal users' money [10]. Trojan.Ransom.Gen utilizes a ransomware threat to block other accesses of injected computers so that the users of injected computers can not gain accesses to their computers. Because this Trojan program blocks the access of infected computers, removing Trojan.Ransom.Gen becomes difficult from injected computers.

Trojan.Rootkit

Trojan.Rootkit can hide files and registry entries through tampering with these files' suffixes [11]. The Trojan program can remove the task manager, disable commands prompt and Registry Editing, deactivate Firefox and security software, pop up a "Blue Screen of Death" screen and make "Log off" button disappear.

Trojan.Spy

Trojan.Spy can monitor the user's operations on an infected computer [12]. This threat can steal user's personal information on hard disks and download the malware to the injected computer. Even Trojan.Spy can send and receive the sensitive files over the network.

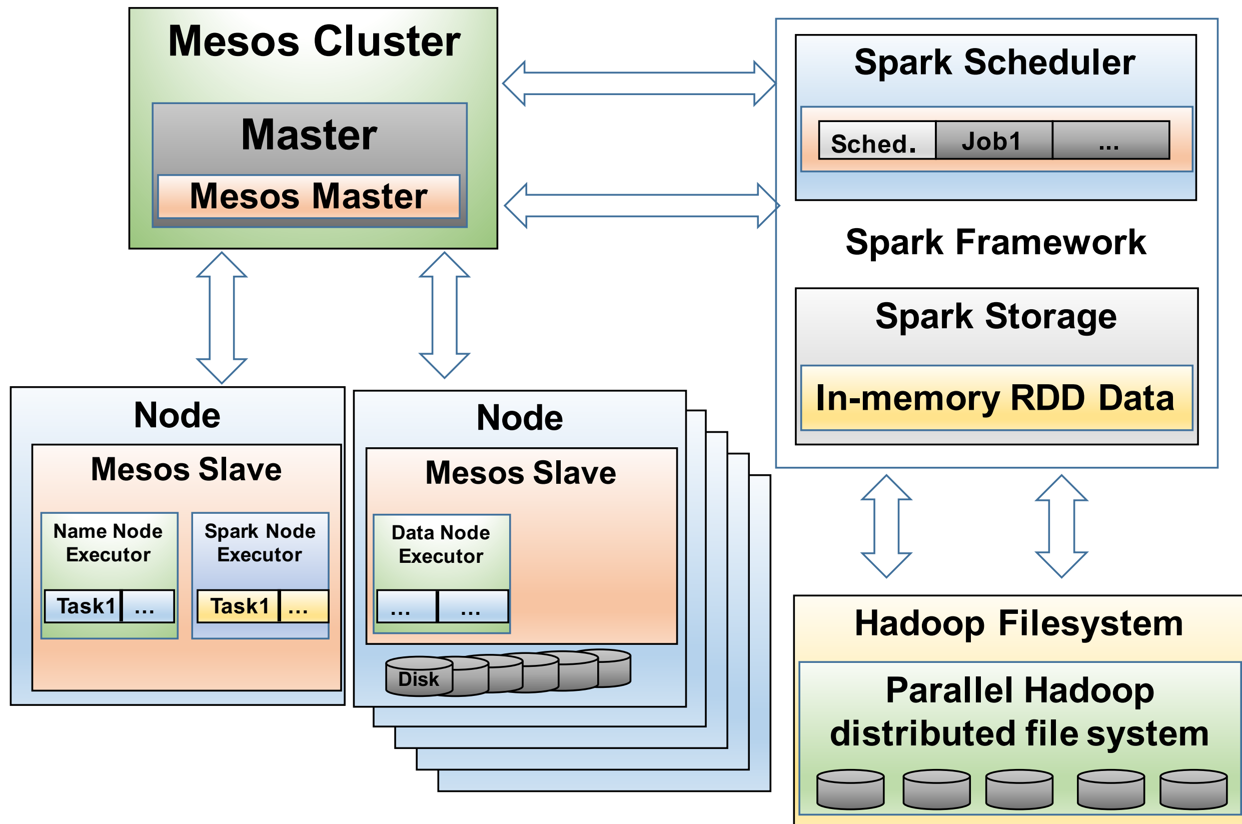

Figure 2.1: Framework of Spark on Mesos

Trojan.ArcBomb

Trojan.ArcBomb [9] attempts to freeze or slow the disk's performance with a large amount of unpacked empty archived data. This program can easily crash a data server with three types of "bomb": archive headers, repeated data, and identical documents of the archive. It is a destructive Trojan to attack the computer and damage the data.

2.2 Apache Spark architecture

Apache Spark [93], as a popular large-scale data processing framework, has been used to improve the performance of iterative machine learning algorithms in parallel. Due to a read-only collection of objects, Resilient Distributed Dataset (RDD), Spark can easily cache the parallel data in memory and iteratively exploit the RDD in the parallel operations, which

eliminates the overhead of I/O communications. Apache Spark is implemented in Scala [8], which is an object-oriented programming language. Moreover, Spark provides the easy-to-use MapReduce-like [33] interfaces to perform parallel computations. In addition, there are pipeline APIs to process the raw data, extract data features, train models and validate results in the prediction procedure [60]. Therefore, Spark efficiently predicts the malicious and benign behaviors of popular applications in Android through large-scale datasets.

Spark provides the RDD to construct a shared dataset for iterative machine learning [60]. Moreover, it supports multiple programming languages [93], e.g., Scala, Java, and R. It integrates other high-level applications and management tools. Spark on Mesos can dynamically partition jobs between Spark and other frameworks, or among the tasks of Spark.

Fig. 2.1 shows the framework of Spark on Mesos. Mesos [47] interpolates a Mesos Master and Mesos slaves to coordinate Spark Scheduler and other applications. When Mesos slaves exist free resources, they report the information to the Mesos Master and the Mesos master then informs the Spark Scheduler. After the Spark Scheduler receives the detailed information, it will decide which job should be launched firstly and send the feedback to Mesos Cluster and Mesos slaves. Spark loads the data into RDDs from Hadoop filesystem (HDFS), which translates parallel disk-based data to in-memory data.

The Apache Spark ecosystem in Fig. 2.2 consists of three layers, Spark optional manager, Spark core and Spark applications. For Spark applications, Spark SQL [15] is used for storing structured data or executes queries via Apache Hive Query Language; Spark Streaming [94] is used to process the live data streams and analyze them; MLlib, as a library [60] is designed for machine learning interfaces; Spark GraphX [89] is implemented to process and manipulate graphics. Spark core provides the mechanism of scheduling, fault tolerance and resource management. Mesos [47], Yarn [85], and Hadoop [77], as the optional managers, contribute to visualize the usage of resources and manage the limited devices.

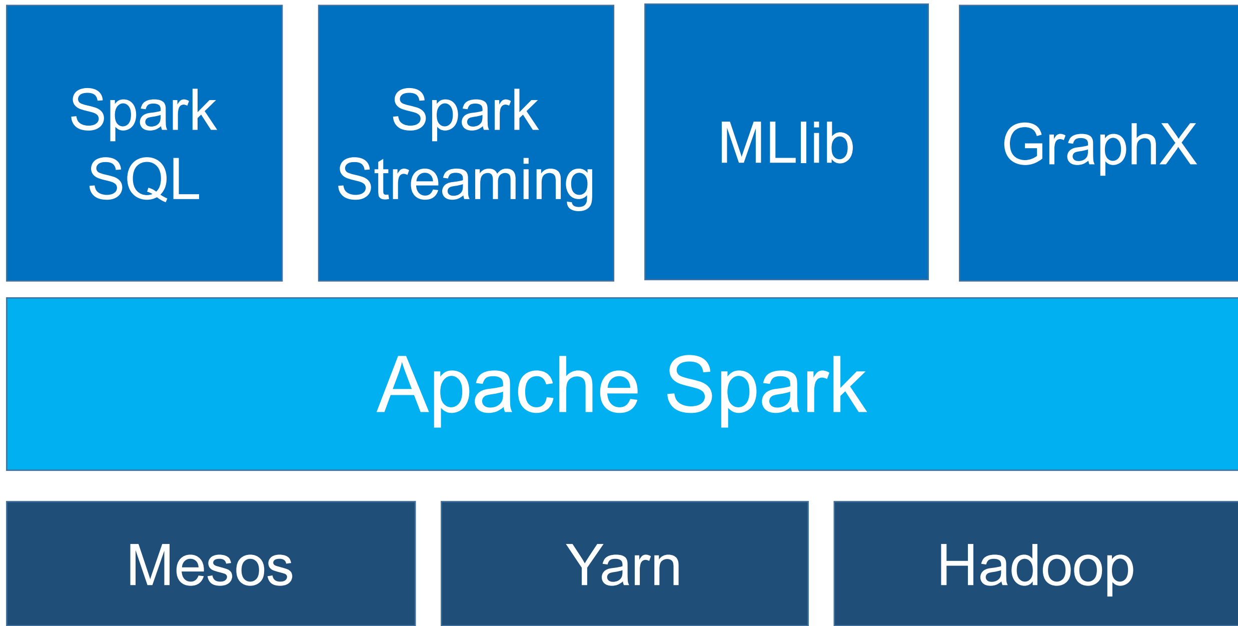

Figure 2.2: Apache Spark Ecosystem

2.3 Artificial Neural Network

In the feedforward Artificial Neural Network (ANN), to minimize the errors between outputs and targets, EBP is generally considered as an efficient training algorithm [67]. Fig. 4.18 shows the architecture of the simplest ANN form and the schematic of the EBP network where they have a layer of inputs, a hidden layer of training neurons, and a layer of outputs. For the simple ANN, the dimension of the input layer equals to the dimension of the original dataset or the reduced dataset. The number of hidden neurons is uncertain due to the characteristic of the different input data. The output layers correlate the desired output which has been known before training an ANN model.

The simple bipolar ANN framework can be expressed by the following equations 2.1 and 2.2:

$$net_j = \sum_{i=1}^{n} x_i w_{ji} + bias_j \tag{2.1}$$

$$output = sign(net) = \begin{cases} 1 & net \geq 0 \\ 0 & net < 0 \end{cases} \tag{2.2}$$

where i, j represents the number of feature patterns from 1 to n and the number of neurons decided by different algorithms (e.g., convolution calculation, adaptive clustering), net_j stands for the j-th neuron's net value and equals to the summation of the product of n patterns and their weights w_{ji} and the bias $bias_j$. Equation 2.2 shows the simple methods of how to determine the output value in output layers. If the net value, *net* from equation 2.1, is greater than or equals to 0, the output value is 1, otherwise, the output value becomes to 0.

2.4 Android PCB kernel structure

The data structure, **task_struct** [59] in process control blocks, as a descriptor of process interaction, has approximately 100 elements to store the information of executing programs. It gives us an elaborate description of a running process, e.g., process state, process priority, scheduling policy, etc., after being allocated by the slab allocator. While measuring the variables constantly invoked by a malware process, some of the noticeable features can be used for delimiting the malicious behaviors. Process control blocks dynamically update and maintain the process identification data, process current state and process control information, so a method of mining PCB in the Linux operating system is proposed to detect and predict malware applications in [65, 76, 84].

To attain the footprint of task management in the data structure (**task_struct**), we implement a software agent to retrieve the element's trace and transform the complicated data structure to a relational data record stored in the local database. Thereby, each data record of the relational database is a tuple with the form:

$\langle hash_key, classifier, task_state, mem_info,\ sche_info, signal_info, others\rangle.$

hash_key The hash table is an efficient data storage and lookup structure that is implemented with a key-value pair. Here hash_key means the unique non- or malware software applications applied in smart telephones. Hash_key denotes the application's name or application's MD5 serial number.

classifier The supervised machine learning algorithms require labeled samples to train a precise classifying model. Classifier represents whether the sample belongs to a malicious app or a benign app. Note that in the experiment **Benign** and **Malware** are their identifiers.

task_state The overview of task execution is defined to describe the exiting case in the task structure. Its return value consists of the special macros to reflect the status of task exiting. Meanwhile, to avoid orphan or zombie processes, the relative signal between parent and children processes is also elemental.

mem_info The traces of memory usage indicate resource demand and process interaction. The data structure generalizes data, code, environment, heap, and stack arguments in detail when a program is executing. It is often referenced by parent and children processes and updated to the latest value by them.

sche_info When the ability of computation of an OS exceeds its threshold, a reasonable scheduling strategy is introduced to increase the system's tolerance. The scheduling information necessitates the system's recovery from a suddenly crashing state. Here, the scheduling information only focuses on the last operation and execution delay.

signal_info The task structure reserves the space for handling received signals. Each process must apply or utilize the limited resources to restrict or make excessive use of CPU, memory, or disk. Moreover, all the threads in the same process share the same signal block. Here the signal information includes the counts of signal variables.

others It conserves the rest of the information of **task_struct**.

In our study, we collect training and testing data sets with 114 features shown in Table 2.1, where 112 features can be used to malware detection and 2 **hash_key** parameters are used to uniquely store data records. In other words, the instance of experimental samples,

the row data record, is a 112-dimension vector. Each dimension (column) represents a feature (a variable in raw data set). From a large amount of multi-dimensional data, the unknown malicious examples are recognized via machine learning algorithms according to the known training dataset.

2.5 112 Android Kernel Variables

The following table 2.1 describes the 112 Android kernel features which are used to detect Android malware. These kernel variables can be classified into 7 categories shown in Section 2.4 and the details are listed in the following table. In order to simplify the variable's names, we assign the related number to the 112 Android features. Among them, the first two variables are useless to detect Android malware. They are the unique identifiers to store the records to database.

Table 2.1: 112 Android Kernel Features

	#	Parameters	Description
hash_key	1	hash	unique apk apps name.
	2	time_instance	time of sampling data.
task_state	3	exit_state	flags of children tasks exiting.
	4	exit_code	a process termination code.
	5	exit_signal	a signal received from exit_notify() function.
	6	pdeath_signal	a signal from dying parent process.
	7	jobctl	reserved to handle siglock.
	8	personality	process execution domain.
	9	maj_flt	major page faults.
	10	min_flt	minor page faults.
	11	arg_end	ending of arguments.
	12	arg_start	beginning of arguments.
	13	end_brk	final address of heap.
	14	start_brk	start address of heap.
	15	cache_hole_size	size of free address space hole.

Continued on next page

Continued from previous page

#	Parameters	Description
16	def_flags	default access flags.
17	start_code	start address of code component.
18	end_code	end address of code component.
19	start_data	start address of data.
20	end_data	end address of data.
21	env_start	start of environment.
22	env_end	end of environment.
23	exec_vm	number of executable pages.
24	faultstamp	global fault stamp.
25	mm_flags	access flags of linear address space.
26	free_area_cache	first address space hole.
27	hiwater_rss	peak of resident set size.
28	hiwater_vm	peak of memory pages.
29	last_interval	last interval time before thrashing.
30	locked_vm	number of locked pages.
31	map_count	number of memory areas.
32	mm_count	primary usage counter.
33	mm_users	address space users.
34	mmap_vmoff	offset of vm files.
35	mmap_base	base of mmap areas.
36	nr_ptes	number of page table entries.
37	pinned_vm	number of pages pinned permanently.
38	reserved_vm	number of reserved pages.
39	shared_vm	number of shared pages.
40	stack_vm	number of pages in stack.
41	total_vm	total number of pages.
42	task_size	size of current task.
43	token_priority	priority of task token.
44	nivcsw	number of in-volunteer context switches.
45	nvcsw	number of volunteer context switches.
46	start_stack	initial stack pointer address.
47	rss_stat_events	used for synchronization threshold.
48	usage_counter	reference count for task_struct of process.
49	nr_dirtied	used in conjunction with nr_dirtied_pause.
50	nr_dirtied_pause	used in conjunction with nr_dirtied_pause.

Continued on next page

Continued from previous page

	#	Parameters	Description
	51	dirty_paused_when	start of a write-and-pause period.
	52	normal_prio	priority without taking RT-inheritance into account.
	53	utime	user time
	54	stime	system time
	55	utimescaled	scaled user time
	56	stimescaled	scaled system time
sche_info	57	last_queue	time when the last queue to run.
	58	pcount	number of times running on the CPUs.
	59	run_delay	time spent on waiting for a running queue.
	60	state	flag of unrunable/runnable/stopped tasks.
	61	on_cpu	flag of locking or unlocking running queue (default 0).
	62	on_rq	flag of migrating a process among running queues.
	63	prio	denotes normal priority (0-99) and realtime (100-140).
	64	static_prio	holds processes initial prio.
	65	rt_priority	Denotes normal priority (0) and realtime (1-99).
	66	policy	scheduling policy used for this process.
	67	rcu_read_lock _nesting	Flag denoting if read copy update is occurring.
	68	stack_canary	Canary value for the -fstack-protector gcc feature.
	69	last_arrival	when last request runs on CPU.
	70	flags	Denotes need to use atomic bitops to access the bits.
	71	ptrace	flag. denotes if ptrace is being used.
signal_info	72	group_exit	flag of group exit in progress.
	73	signal_nr_threads	denotes number of threads.
	74	signal_notify_count	compared with count. If equal, group_exit_task is notified.
	75	signal_flags	used as support for thread group stop as well as overload of group_exit_code.
	76	signal_leader	boolean value for session group leader.
	77	signal_utime	same as task_struct but used as cumulative resource counter.
	78	signal_cutime	cumulative user time.
	79	signal_stime	used as cumulative resource counter.
	80	signal_cstime	Cumulative system time.
	81	signal_gtime	Group time. Cumulative resource counter.
	82	signal_cgtime	Cumulative group time. Cumulative resource counter.
	83	signal_nvcsw	used as cumulative resource counter.
	84	signal_nivcsw	used as cumulative resource counter.
	85	signal_cnvcsw	Cumulative nvcsw.
	86	signal_cnivcsw	Cumulative nivcsw.

Continued on next page

Continued from previous page

	#	Parameters	Description
	87	signal_maj_flt	used as cumulative resource counter.
	88	signal_cmaj_flt	Cumulative maj_flt.
	89	signal_cmin_flt	Cumulative min_flt.
	90	signal_inblock	Cumulative resource counter.
	91	signal_oublock	Cumulative resource counter.
	92	signal_cinblock	cumulative inblock.
	93	signal_coublock	Cumulative oublock. Cumulative resource counter.
	94	signal_maxrss	Denotes memory usage. Cumulative resource counter.
	95	signal_cmaxrss	Denotes cumulative maxrss. Cumulative resource counter.
	96	signal_sum_sched _runtime	Cumulative schedule CPU time.
	97	signal_audit_tty	Denoted status of audit event resulting from tty input.
	98	signal_oom_score _adj	Denoted status of audit event resulting from tty input.
	99	signal_oom_score _adj_min	minimum.
	100	sas_ss_sp	signal handler pointer.
	101	sas_ss_size	size of signal handler pointer.
others	102	gtime	guest time
	103	link_count	number of symbolic links
	104	total_link_count	total number of symbolic links.
	105	sessionid	process session ID
	106	parent_exec_id	execution domain belonging to parent thread ID.
	107	self_exec_id	execution domain belonging to self thread.
	108	ptrace_message	result block of ptrace messages.
	109	timer_slack_ns	Used to round out poll() and select() etc timeout values. Value is in nanoseconds.
	110	default_timer_slack_ns	Same as timer_slack_ns.
	111	curr_ret_stack	index of current stored address in ret_stack.
	112	trace	state flags for use by tracer.
	113	trace_recursion	bitmask and counter of trace recursion.
	114	plist_node_prio	priority value belonging to a node on a plist.

Concluded

2.6 Brief Summary of Previous Malware Detection through Behavior

In general, malware detection falls into a plethora of categories based on different classification methodology. Kim *et al.* [50] proposed power-aware malware detection by collecting power consumption samples and calculating the Chi-distance. Afterwards, Liu *et al.* [58]

designed the state machine matrix to collect power consumption data and use machine learning algorithms to identify malware apps. Behavior-based analysis for malware detection was proposed by Shabtai *et al.* [74], where they offer a high level framework of malware detection, including feature selection and the number of top features. However, Shabtai *et al.* did not evaluate what kind of features should be selected and how many of them could be used to detect malware.

Rastogi *et al.* [64] researched the anti-malware software and provided a method, named DroidChameleon, which listened to the system-wide message broadcast and compared their footprints with a single rule. In addition, Lanzi *et al.* [55] proposed a system-centric model of performing a large-scale data collection of call sequence and training the data with n-gram methods. Demme *et al.* [34] proposed a machine learning based detection technique with performance counter. They analyzed the feasibility of online malware detector and came up with a tentative plan of hardware implementation of malware detector. Shahzad *et al.* [76] proposed a dynamic malware detection technique in the Linux system, in which they acquired a short list of Linux kernel features to train their model of machine learning. Nevertheless, these techniques just focus on collecting history traces with low dimension, where data sets from kernel or other applications contain few features.

Chapter 3

Problem Statement

In this chapter, we summarize the problems of Android malware detection, including the brief review of the Android architecture, problems for dynamic Android malware detection and static Android malware detection. Additionally, we show the challenges in TstructDroid and our goals for Android malware detection. Furthermore, the problems for in-memory large-scale data training and artificial neural network are discussed in this chapter as well.

3.1 Android Architecture

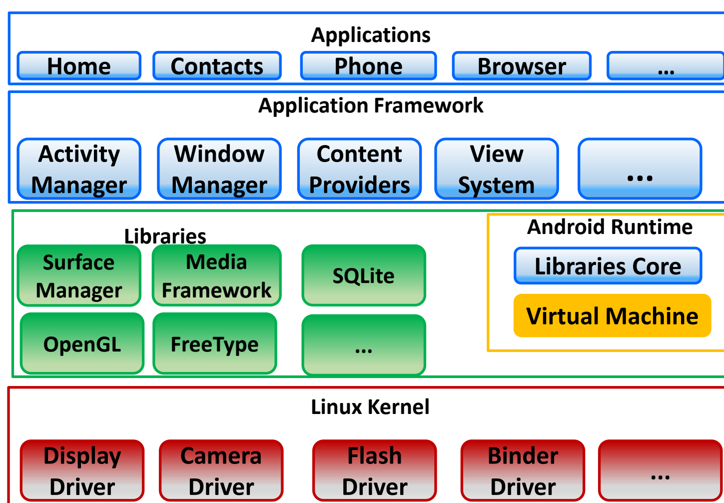

Figure 3.1: Android Architecture

Fig. 3.1 shows the components of Android system [80] is comprised of Applications, Application Framework, Libraries & Android Runtime, Linux Kernel. The application layer

is located on the top of the Android system, with the responsibility for installation and operation of the user software, e.g., mail, browser, or music, etc. The application framework contains the high-level services in the form of java classes for the communication between the application layer and the Android libraries. The Android libraries layer provides the resource access from the second layer, in addition to those C/C++ based applications. The Android runtime encompasses two important components, core libraries for the standard java language and light-weight Android virtual machine. The bottom layer, the Linux kernel, is the core of Android architecture, which handles the process scheduling, memory management, power management, communication between hardware and software, etc.

In this open-source software platform, our data collection mainly focuses on the bottom layer, the Linux kernel, where task_struct [59] elaborates process interaction, memory usage, signal utilization and the information of other resources. The data structure, task_struct, contains 112 features of executing programs which can be used to detect Android malware.

3.2 Dynamic Android Malware Detection in Linux Kernel Layer

There are a lot of Android malware detection techniques using machine learning classifiers. Schmidt et al. described a malware detection mechanism from Linux kernel perspective [71]. They came up with an Event Detection Module (EDM) to extract the Android kernel features and attempted to use machine learning technique to classify the malicious apps. However, they did not implement the EDM to further obtain the training model for malware detection. In our study, we have completed the data collector with the same functionality as EDM and trained the machine learning models for malware detection. Bläsing et al. proposed a sandbox to print the kernel information for static and dynamic analysis [23], but did not employ the technique to Android malware detection in practice.

The " Andromaly " framework [74] also collected the Android kernel features to find the best combination using machine learning methods. They ranked those features with the different dimensional reduction methods and achieved top 10 features that outperformed the

other combinations. Among those 10 features, there were 4 memory-related features with the highest ranks. A Multi-level Anomaly Detector for Android Malware, MADAM [35], could monitor Android activities at the kernel level and the application level to detect malicious intrusion with machine learning techniques. However, only 4 kernel features were monitored.

In [46], Ham et al. collected 32 resource features of network, SMS, CPU, power, memory, virtual memory and process. The random forest classifier achieved the best performance for 35 Android applications and the features of memory and virtual memory were appropriate to accurate classification. B. Amos et al. presented a STREAM framework to rapidly validate mobile malware machine learning classifiers [14]. Only 30 similar kernel features were collected, including process, CPU, memory and network considering few application resource constraints.

In [75], F. Shahzad et al. proposed a TstructDroid framework to discriminate Android benign and malicious apps. They gathered the dataset consisting of 110 malicious apps and 110 benign apps for 32 Android kernel features. Due to the difficulty in training a suitable machine learning model with the relatively large dataset, they used the techniques, Discrete Cosine Transform and Cumulative Variance to detect the small changes in kernel features. In fact, the two methods introduced the overfitting issue when training a suitable model instead of improving the accuracy rate.

T. Isohara et al. [49] designed an audit application called logcat on virtual machine to monitor the application behaviors and proposed a kernel-based behavior analysis to inspect the Android malware. However, they only collected 2 types of system logs, process management and file I/O. Due to the lack of empirical evaluation, the offline analysis of log data **might** detect the Android malware via the pattern matching in large log files.

3.3 Static Android Malware Detection in Other Layers

Static analysis of executable binaries has been applied to Android malware detection [70]. In [19], L. Batyuk et al. proposed a static analysis method for automated

binary assessment and malicious event mitigation, which depended on the open-source decompilation tools to decode binary applications to initial forms. A. Shabtai et al. [73] further investigated the code of Android applications and evaluated XML-based features with dimensional reduction methods and machine learning classifiers.

Yerima et al. implemented an automated tool to reverse Android applications for collecting the useful features and evaluated the Bayesian classifiers [90]. To select the most relevant features, 58 Android application properties were ranked with dimension reduction methods. It waw discovered that 15 to 20 features were enough to detect Android malware. ComDroid [30] analyzed the intents of Android applications to discover the vulnerabilities in Android system. Similarly, DroidMat [87] extracted the features of permissions and intents and used the K-means method to recognize Android malware. DroidChecker [28], as an Android malware detection tool, used the interprocedural control flow to find the capability leaks in Android applications. Other methods, e.g., FlowDroid [17], ProfileDroid [86], RiskRanker [44], ScanDal [51], AndroidLeaks [43], also statically analyzed the information to detect Android malware. Our method can detect the Android malicious applications when they are executing. DroidAPIMiner [13] extracted the API-level features in Android and statically evaluated the data samples with different classifers. Additionally, the authors analyzed the frequently invoked features in API calls to gain the Android malware behavior.

J. Sahs et al. gathered the permission features with an open-source tool and discriminated the Android malicious and benign applications after refining them with control flow graphs in [68]. Y. Zhou et al. proposed a scheme, permission-based behavioral footprinting [96], to detect malicious applications in official and unofficial Android markets. I. Burguera et al. proposed Crowdroid [27], a behavior-based malware detection framework, which built the dataset with behavior system call feature vector. To leverage a Hidden Markov Model to predict Android malware, L. Xie et al [88] designed a system named pBMDS, which employed a statistical method to learn the malware behaviors in cellphone devices. Schmidt et al. propsed an approach to collaborative Android malware detection with the

static analysis of executables in [69]. D. Barrera et al. explored permission-based models for Android malware detection with Self-Organizing Map algorithm in [18]

3.4 Challenges in TstructDroid and Our Goals

In TstructDroid [75], a cumulative Variance Based Detection (VBD) technique with Android kernel features has been proposed to analyze Android malware. To build a realworld dataset. the framework tests 110 malicious and 110 benign Android applications from the Android marketplace. In consideration of the large feature dataset, it is difficult to discriminate malware and benign applications with the entire data samples. Therefore, cumulative variance of frequency of kernel features obtained with Discrete Cosine Transformation (DCT) is used to detect Android malware. Moreover, the VBD method uses 32 Android kernel features and the decision tree classifier.

TstructDroid presents the detailed procedure to reduce the large dataset. The first step is that DCT transforms values of kernel features to frequencies. After applying DCT, the cumulative variance is further used to reduce the data size. However, these methods degrades the classification performance because of the loss of the original data similarity. Without changing the dimension of kernel features, DCT can discard the important kernel information for lossy compresses of data points. In addition, the cumulative variance does not reduce the data size of Android features, therefore, it can not speed up the classification. Moreover, the cumulative variance introduces the extra loss of data integrity.

Currently, the Android kernel task structure, **task_struct**, includes 112 features which has added 23 new features since 2013. TstructDroid, analyzing 32 kernel features, does not show whether other features are important to Android malware detection. They can not prove the 32 kernel features are relevant to malware detection. Some features has disappeared in the current Android system due to system upgrade. Therefore, a comprehensive analysis of the entire kernel features would be helpful to predict the trend of kernel features modified by attackers.

In our study, we collect 112 latest Android kernel features to construct an accurate dataset. To remove the redundant records in the original dataset, we apply the dimensional reduction techniques to rank these features. Furthermore, we design the clustering method to reduce the data size instead of DCT transformation and cumulative variance calculation.

3.5 In-Memory Large-Scale Data Training

Currently, the Android phones have a rapid growth accompanied by the rise of malicious threats. In terms of the severe damage inflicted by malicious apps, Android attackers regularly attempt to steal user private information, obtain administrator privilege, or misuse resources. Recently, Kaspersky Lab reported that the proportion of malicious attacks in 2015 for Android software increased by a factor of two in Trojan Banking malware families [42]. Consequently, a myriad of malware detection techniques [26, 49, 95, 55, 31] have been proposed to address this issue and safeguard Android systems. Among them, kernel-based detection [49] has grown in popularity because this approach can audit all applications of an Android phone and obtain detailed log information from a Linux[1] kernel layer. However, the size of data collection of kernel parameters increases dramatically due to scanning the whole kernel structure (15,000 records / 20 s) while acquiring a training dataset. After collecting the large-scale dataset with 112 features from more than 1,000 Android applications, the local computer can not deal with such huge data samples in time.

Obviously, it is difficult to train a good model using a large-scale dataset for malware detection because of the limitation of memory and CPU. Especially, the memory usage becomes a bottleneck for improving the accuracy of classification and reducing the training cost. On the other hand, frequent operations of disk I/O read and write cause performance degradation and increase the extra overhead while training the detection model. Take an example, when we use a 6MB dataset to train the Android detection model, the process of dealing with such dataset needs several hours. In order to shorten the training time

[1] Linux 14.04.1-Ubuntu

and enlarge the memory size, we aim to provide a parallel malware detection framework to analyze and evaluate Android datasets. A Spark-based malware detection framework is presented to preserve the prediction performance and reduce the cost of disk I/O.

3.6 Artificial Neural Network

Artificial Neural Network (ANN) approaches can detect the malware based on the techniques of machine learning through training linear or nonlinear classification models. ANN successfully recognizes the malicious intrusion through feature selection and analysis of critical infrastructures [57, 32, 81]. The advantage of ANN detection is that ANN approaches can capture more characteristics from the undisciplined data samples and lead to a good classification result [36]. However, its main drawback is that ANN approaches can not train a precise model for the large-scale dataset, even reduce the classification performance. Among ANN techniques, the EBP (Error Back Propagation) algorithm has been mostly utilized to solve the issues of classification and approximation [67]. However, in terms of resource demand, the consumption of training an EBP model is high and the accuracy performance is not always global optimal. In contrast, the RBF (Radial Basis Function) network can have a faster training speed and a higher accuracy performance [61] while training the large-scale data samples.

Compared to the traditional ANN approaches, a RBF network-based Android malware detection method can improve the performance in which the heuristic approach of clustering, K-means algorithm, calculate and select the centers of the large dataset. The clustering method will calculate the Euclidean distance among the large amount of data points which measures the similarity of malicious or benign samples. The RBF hidden centers can be dynamically selected by a heuristic approach and the large-scale datasets of 2550 Android apps are gathered by our automatic data sample collector. We design and implement the algorithms of the RBF network and the Error Back Propagation (EBP) network.

Chapter 4

System Design

In this chapter, we introduce our designs for Android malware detection and present an automatic data collector design in our study, including local machine learning techniques, parallel malware detection techniques, and RBF network-based malware detection.

4.1 Overview of Multiple Kernel Features

To differentiate malware applications from benign applications in the Android market, we have gathered Android information of the kernel block which is similar to the Linux kernel parameters in PCB via cellphones. These samples of parameters of Android applications reflect the changes of CPU and memory when malicious apps attempt to steal critical information of administrators or normal users. With our collection of 550 Android applications, where each app contains 15,000 data records composed of 112 kernel parameters, scanning the entire file to locate the analogical attributes is unfeasible manually. Furthermore, each original data record or reduced data record includes the high dimensional features. Therefore, choosing a good subset of these features influences the detection results of out-of-sample malicious data. Typically, short sampling lists ignore the hidden characteristics of learning data sets.

4.1.1 Overview of Malware Behavior in Kernel Level

Different malware apps can be injected in different layers of operating systems (e.g., application layer, kernel image layer, BIOS layer or CPU layers [78]). Potentially, to steal significant information, such as user passwords or bank account data, attackers inject a virus in the application level of Android systems by masquerading themselves to useful software

or applications. Moreover, a Trojan horse fakes the authentic application to persuade users to install it, so that the intruder can control a user's next operation in his cellphone. In existing user mode of Android systems, it is difficult to dig out anomalous behavior from good processes since the attacker hides the modification in other applications. A common and profitable technique to perceive the malicious intrusion is to capture the process's exceptions in the kernel module during the systemś execution.

While malicious software is running as a regular program, some programs are altered with the adjustment of the kernel's task augments, particularly physical or virtual memory usage. In total, there are more than 30 parameters of memory usage to facilitate the memory manipulation in the Linux kernel code. 20 percent of the total parameters, with obvious behavioral footprints, is helpful to detect malware. In this work, we use 112 kernel parameters of tasks and processes for the behavior-based malware program detection.

With more bizarre behaviors at the kernel level, malicious programs attempt to grab more interaction resources (e.g., CPU, memory, disk, or system calls [55, 76]), for obstructing other normal programs. From the task structure footprint collected in our database, the information of a memory usage of the current process are referenced and modiified more frequently than the others. The attributes of the active memory structure can aid defenders to detect these well-camouflaged malicious processes in Android applications, since the behavior models in terms of memory features are different between malware and nonmalware systems. Retrieving the active process's footprint, e.g., pages swapping, pages mapping or pages sharing, is significant in learning kernel behaviors.

4.1.2 A Case Study of Features in Malware Vs. Goodware Distribution

To illustrate the importance of relative kernel parameters, the scatter distribution of malware and goodware apps is shown in the figures (Fig. 4.1, Fig. 4.2, Fig. 4.3, and Fig. 4.4), where $Shared_vm$, $Total_vm$ and $Signal_nvcsw$ stand for the number of shared pages or files of memory mapping address of a process, the total number of memory pages utilized by all

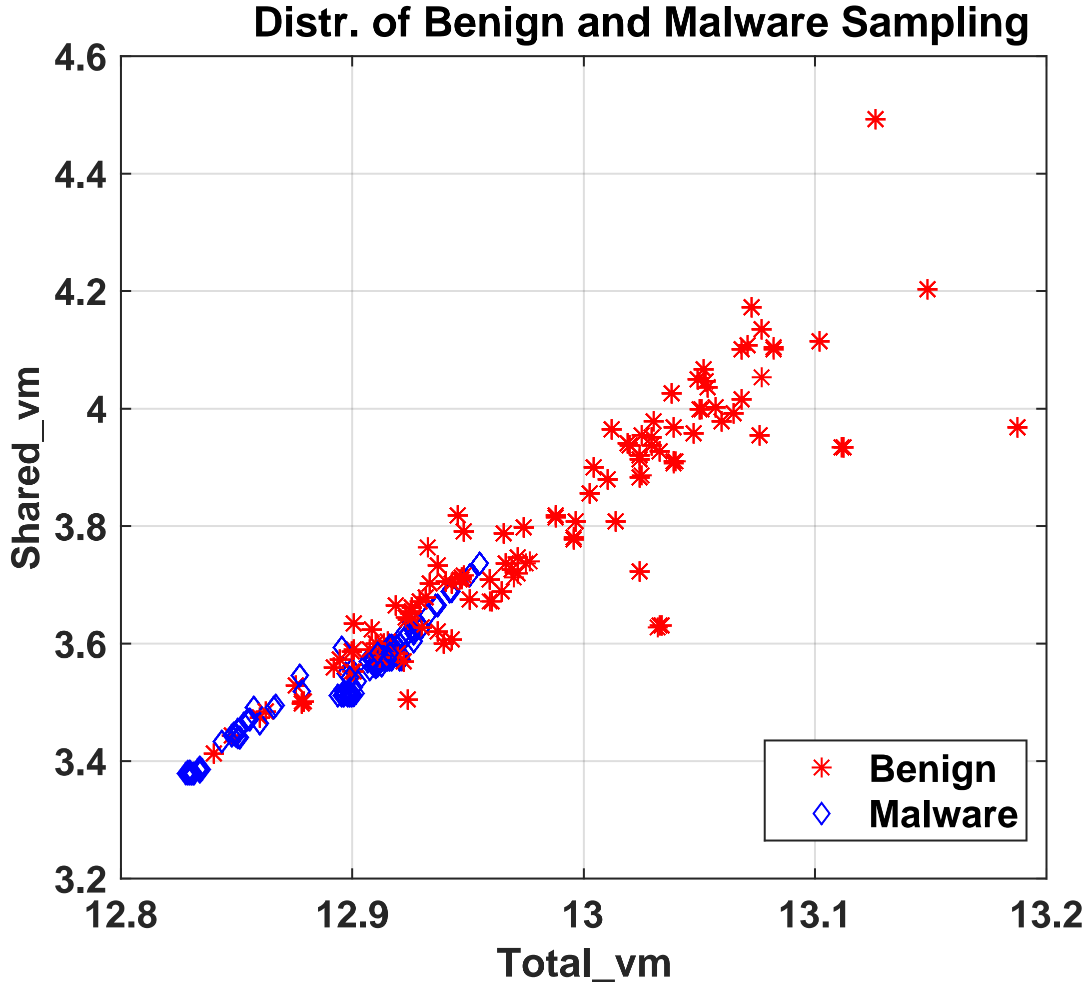

Figure 4.1: 2D Distribution of 100 benign and 100 malware samples with Shared_vm and Total_vm

VMA regions in the current process, and the number of volunteer context switches a process makes, respectively.

Fig. 4.1 shows the 2D distribution of 100 benign and 100 malware samples, and each sample contains 10 instances in discrete time points. The benign samples are mainly distributed in the top-right area comparing the malware locating in the bottom-left quarter. However, among all the instances, there are a large amount of benign samples overlapping with the malware at the range: $Total_vm \in [12.9, 13], Shared_vm \in [3.5, 3.7]$. To classify the two applications precisely, more behavior-based features, in terms of kernel parameters, have to be introduced for correctly transforming them to a multi-dimensional space.

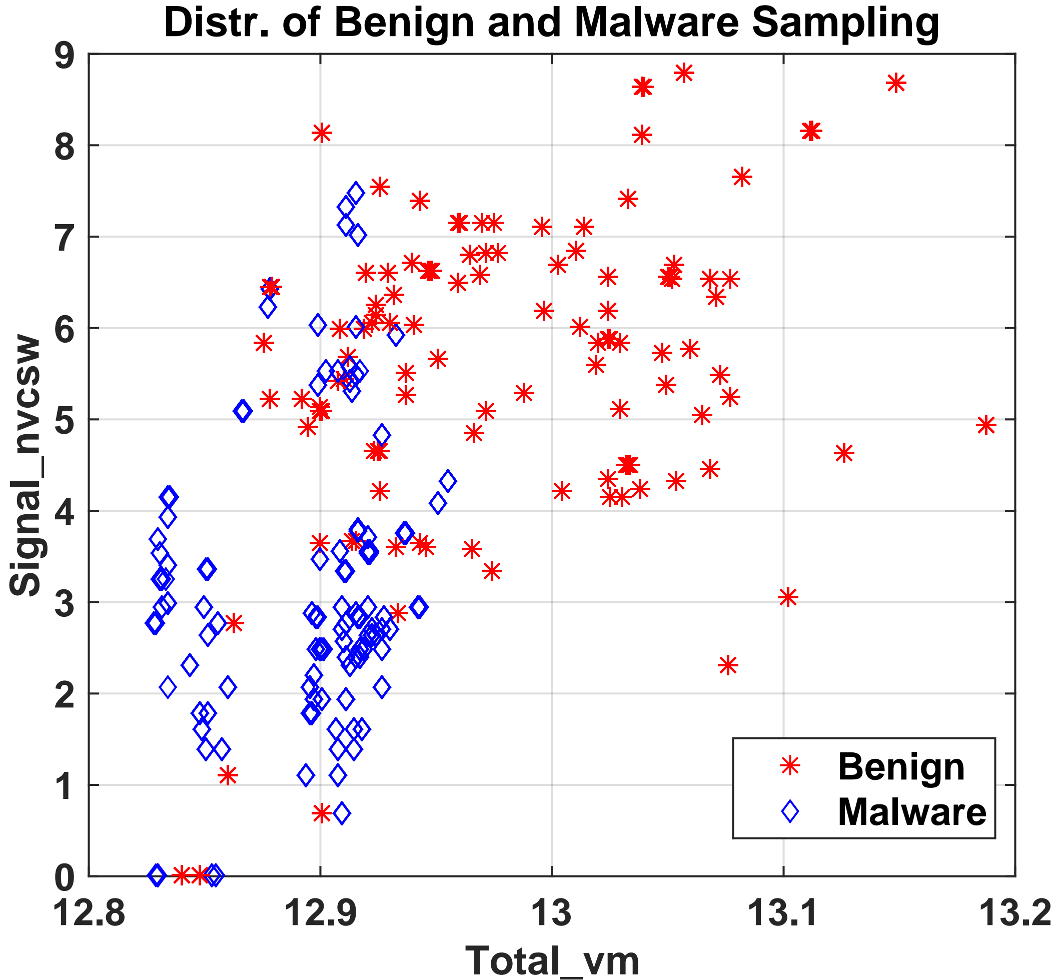

Figure 4.2: 2D Distribution of 100 benign and 100 malware samples with Signal_nvcsw and Total_vm

From the distribution in terms of the *Total_vm* horizontal axis and the *Signal_nvcsw* vertical axis in Fig. 4.2, the benign and malware samples aggregate at upper-right and lower-left areas, respectively. Furthermore, the non-overlapping fields also apparently present the discipline of goodware and malware binary classification. In Fig. 4.3, the malware scatters formally together in the similar position as Fig. 4.1 and Fig. 4.2. Seemingly, malware samples can be differentiated from the goodmalware via *Signal_nvcsw* and *Total_vm* of Fig. 4.2 or *Signal_nvcsw* and *Shared_vm* of Fig. 4.3.

Should we add more kernel parameters to classify these samples? To answer this question, 3 combined parameters are examined and shown in Fig. 4.4. The majority of malware

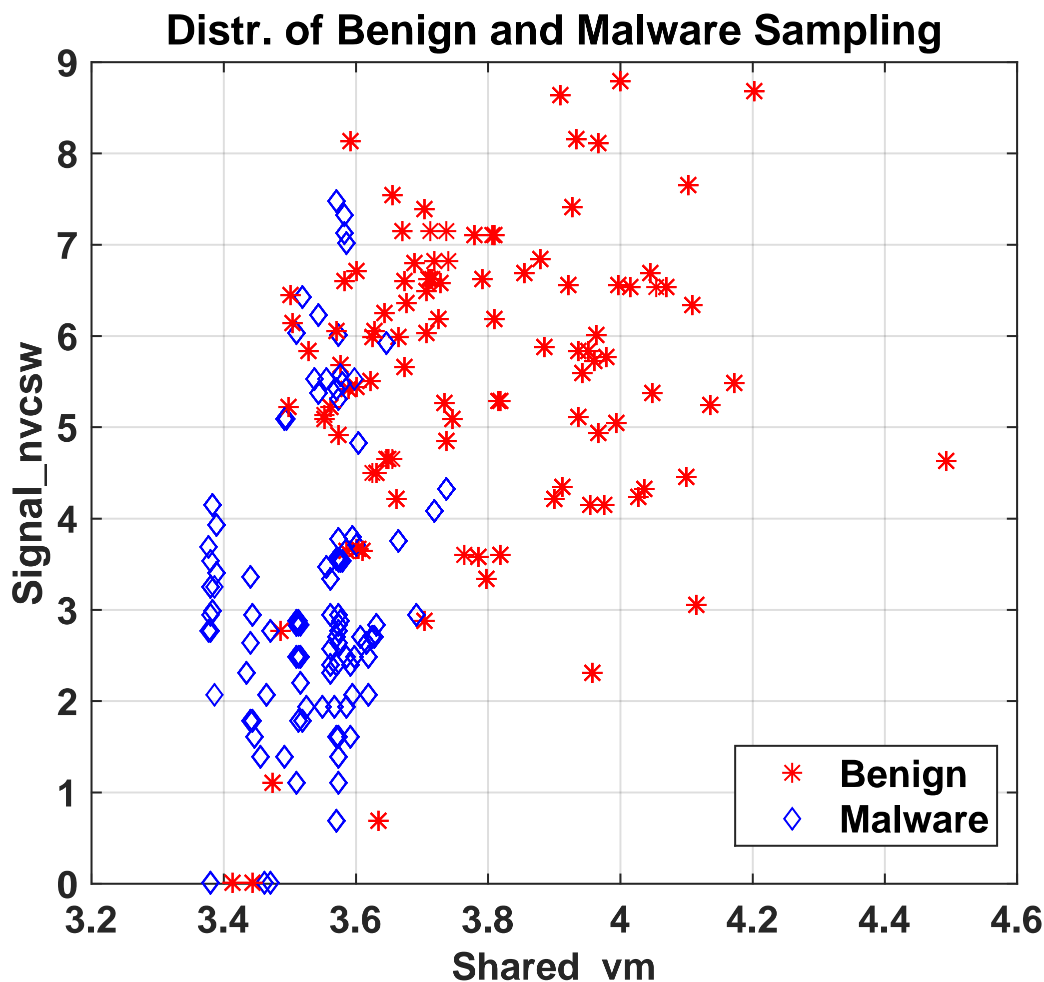

Figure 4.3: 2D Distribution of 100 benign and 100 malware samples with Signal_nvcsw and Shared_vm

samples are isolated to the local space, which improves the probability of separating them in multiple dimensions and extracts abnormal behavior's samples. These 3 features are sufficient to classify the two kinds of samples based on our results under the situation where the customer does not require an exceedingly high accurate rate of classification (commonly the rate is lower than 90%). From the 3D view, we also found samples disobeyed straightforward arithmetical distribution since there exist few malware scatters mixing with the benign. In fact, to utilize more process' features is in favor of identifying the malware in Android platforms.

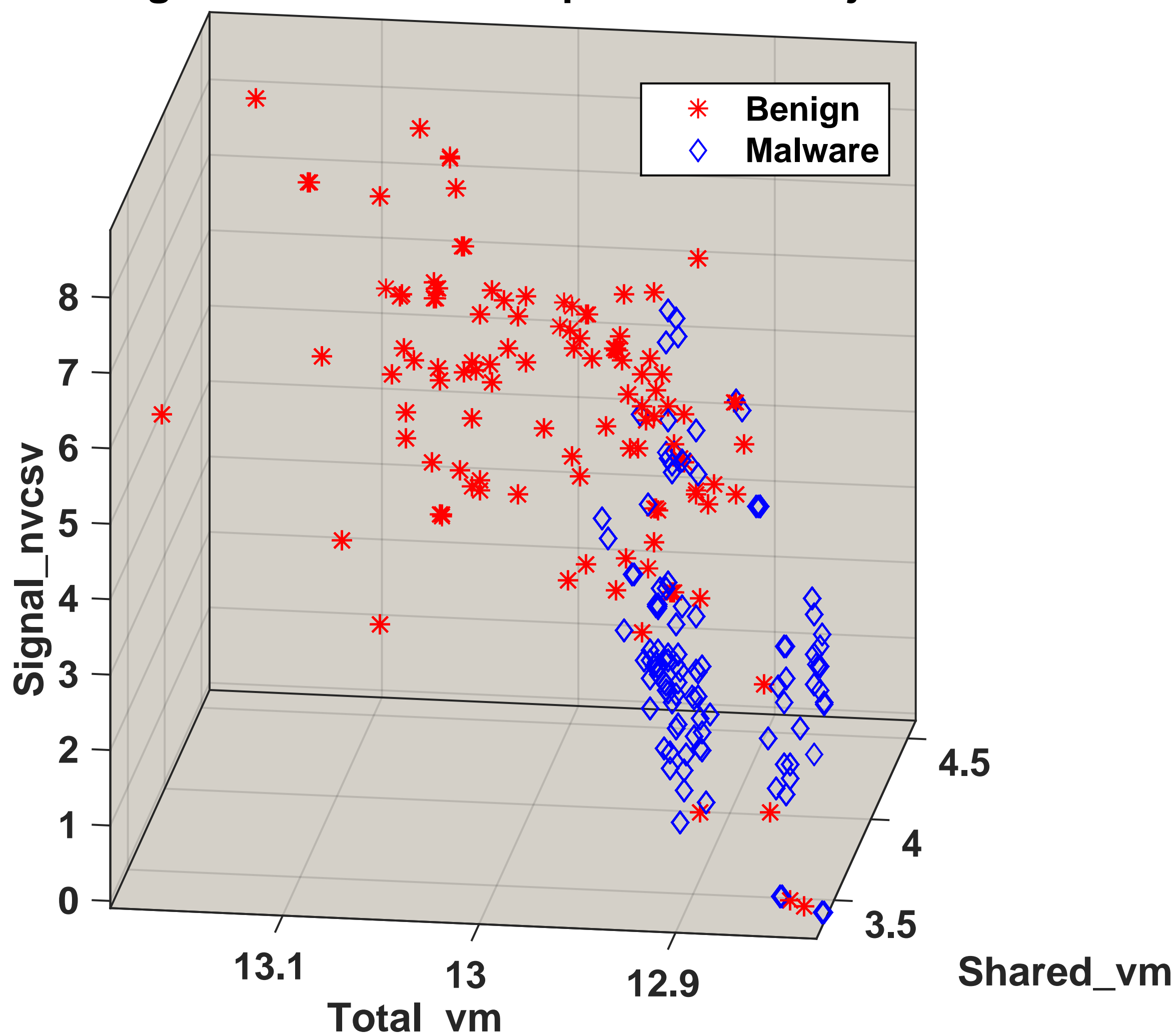

Figure 4.4: 3D Distribution of 100 benign and 100 malware samples: with the increase of the number of dimensions, benign and malware samples cluster together in different areas.

4.1.3 Measurements of Multiple Dimensional Kernel Features

The data structure, task_struct [59] in process control blocks, as the descriptor of process interaction, has approximately 100 sub-structures to store the information of executing program. It gives us an elaborate description of a running process, e.g. process's state, process priority, scheduling policy, etc., after allocated by the slab allocator. While measuring the variables constantly invoked by a malware process, some of noticeable features can be used for delimiting the malicious behaviors. Process control blocks dynamically update and

Table 4.1: Key Variables of Active Processes

#	Parameter	Description
(a)	total_vm	total number of memory pages over all VMAs
(b)	exec_vm	number of executable memory pages
(c)	reserved_vm	number of reserved memory pages
(d)	shared_vm	number of shared memory pages
(e)	map_count	number of memory mapping areas
(f)	hiwater_rss	high water mark of resident set size
(g)	nivcsw	number of in-volunteer context switches
(h)	nvcsw	number of volunteer context switches
(i)	maj_flt	number of major page faults
(j)	nr_ptes	number of page table entries
(k)	signal_nvcsw	number of signals of volunteer context switches
(l)	stime	time elapsed in system mode

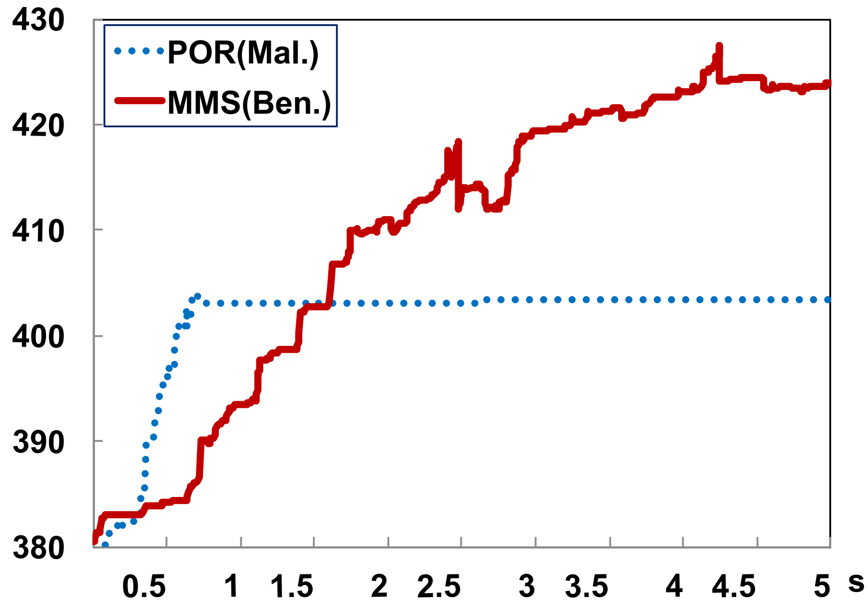

Figure 4.5: total_vm

maintain process identification data, process current state and process control information, so the method of mining PCB in Linux operating system was proposed to detect and predict malware applications in [65, 76, 84]. But the researcher calculated cumulative variance in different time windows after discrete cosine transformation of each instance's value, which introduced more noises and errors if choosing an inappropriate window size in time series.

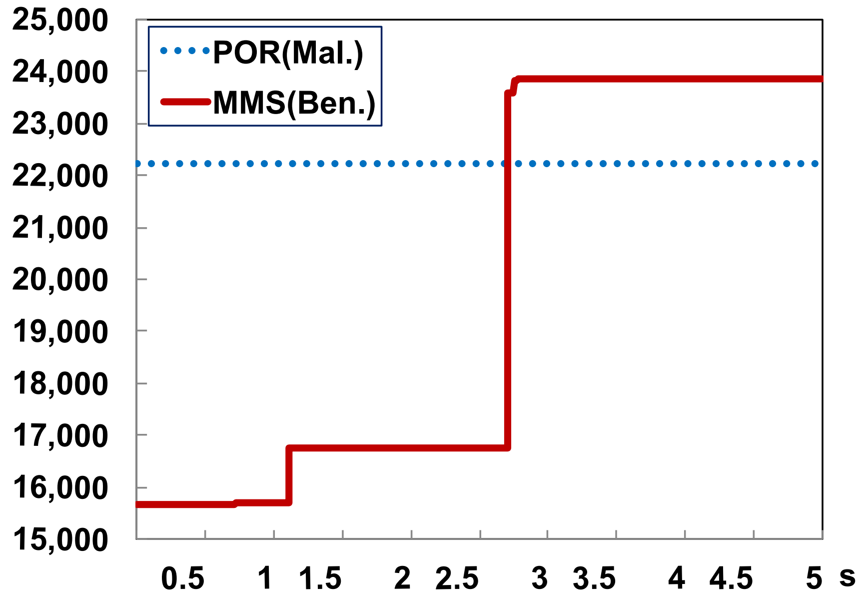

Figure 4.6: exec_vm

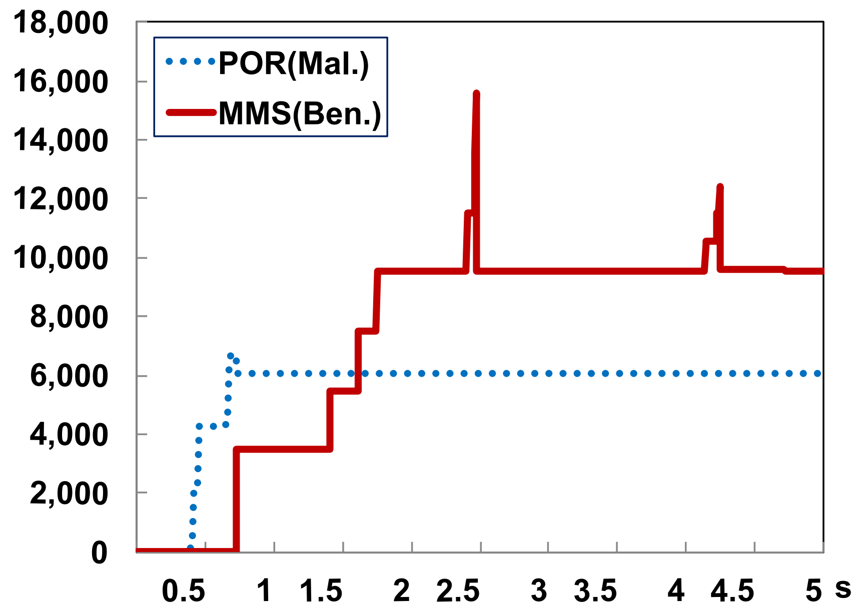

Figure 4.7: reserved_vm

Multi-dimensional datasets have more challenges in mathematical statistics and analysis in terms of data collection and storage capability [41]. As a matter of fact, not all the collected features are useful for distinguishing the malicious software under many cases. In our work, the dimension of the data is also reduced to a suitably short list for performance

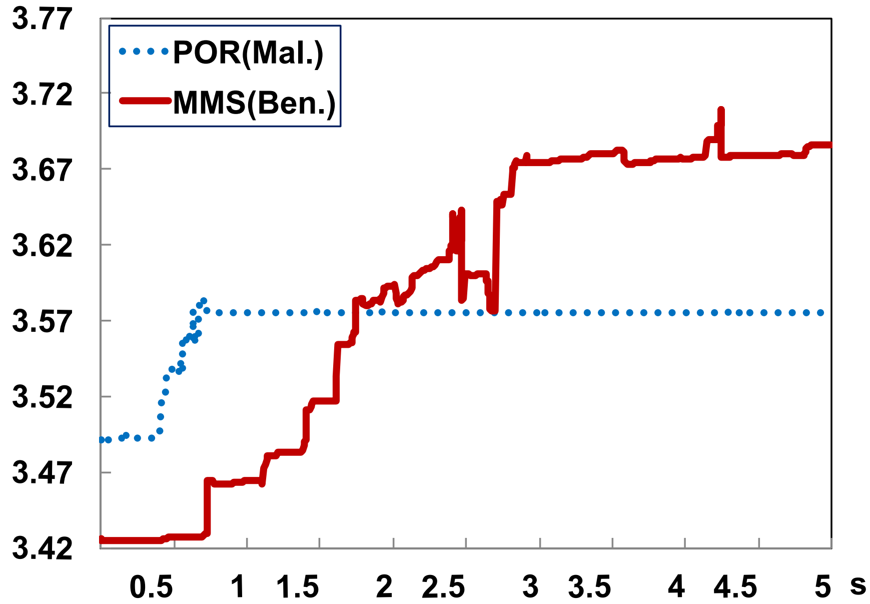

Figure 4.8: shared_vm

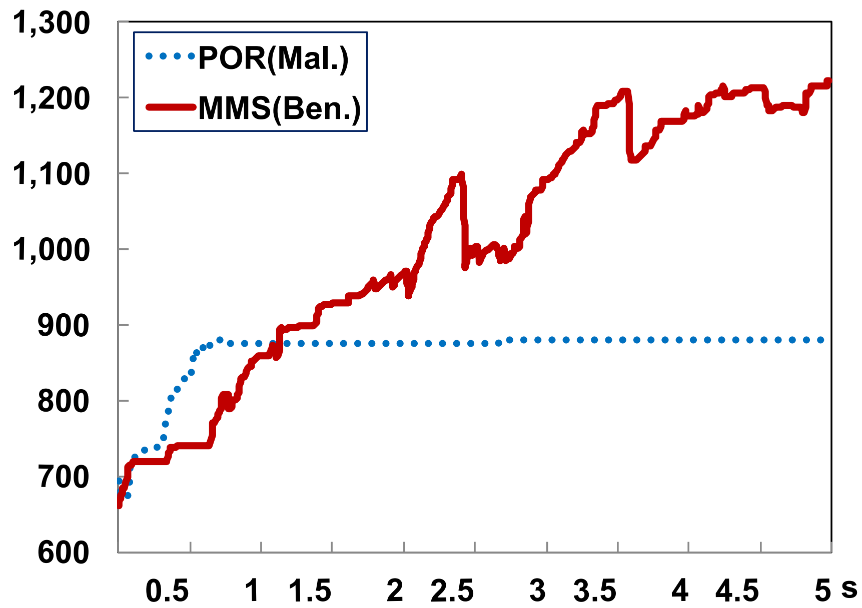

Figure 4.9: map_count

improvement. Meanwhile, we also measure the key process' features and analyze their phenomena illustrated in Table 4.1, Fig. 4.5, Fig. 4.6, Fig. 4.7, Fig. 4.8, Fig. 4.9, Fig 4.10, Fig. 4.11, Fig 4.12, Fig. 4.13, Fig. 4.14, Fig. 4.15, and Fig. 4.16.

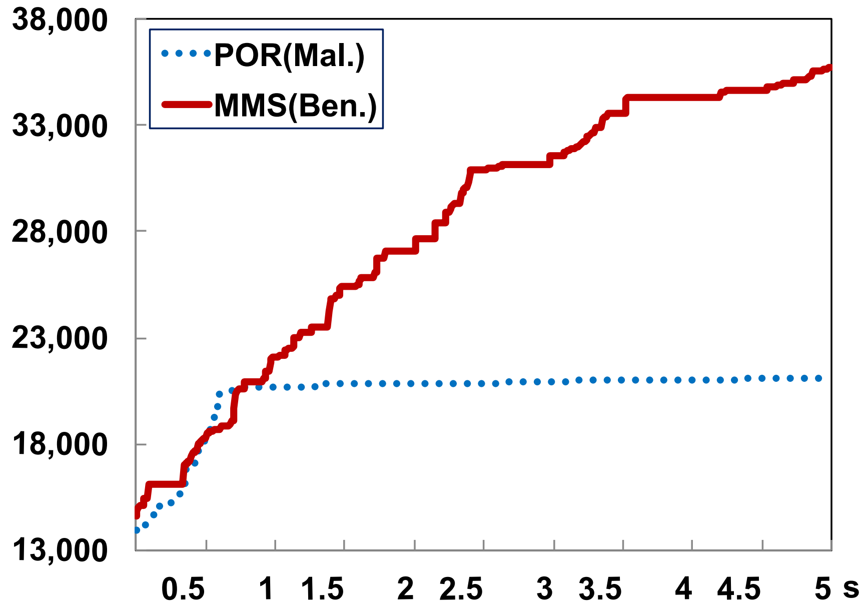

Figure 4.10: hitwater_rss

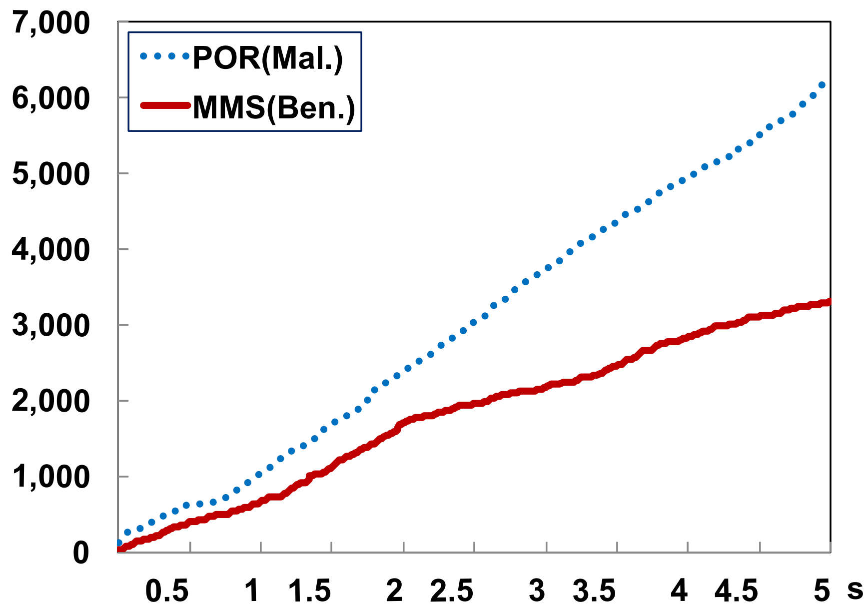

Figure 4.11: nivcsw

Table 4.1 shows the names and descriptions of 12 key parameters of **tast_struct** in PCB. In practice, we collected all the variables of process structure in Table 5 [1] and classified the two kinds with them, including 48 parameters of memory usages, 30 parameters of auditing signals, 15 parameters of scheduling information, 6 **task_state** parameters and 13

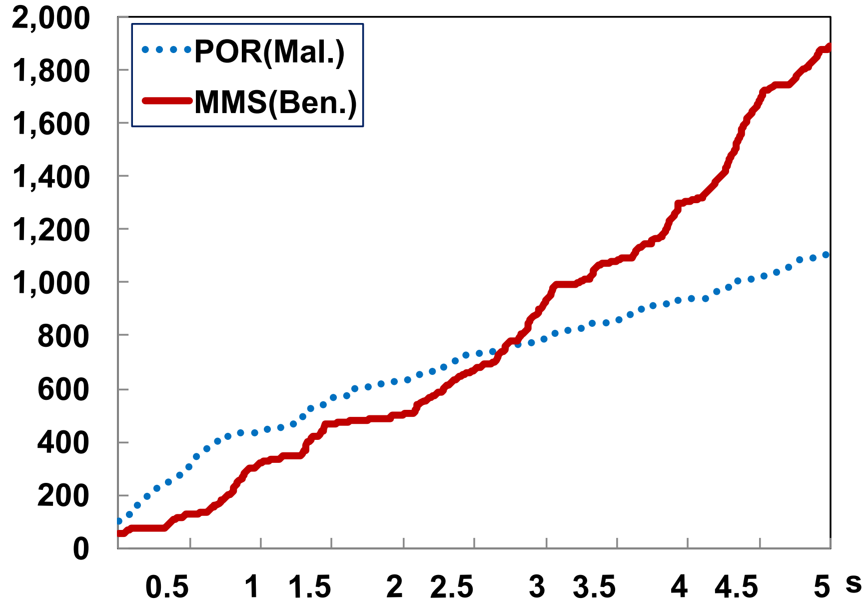

Figure 4.12: nvcsw

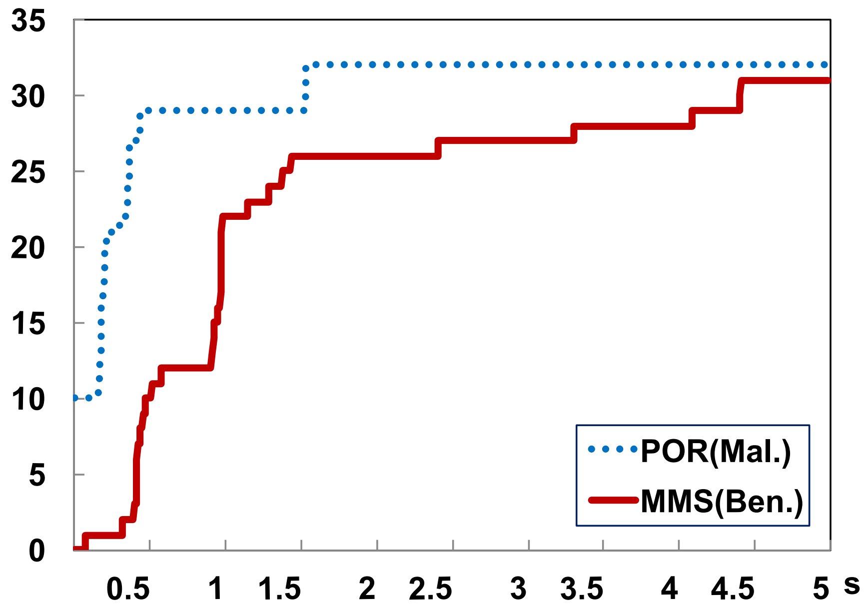

Figure 4.13: maj_flt

parameters of other information on context switches and parent process execution. Theoretically, more dimensions of selected parameters among the total 112 parameters gain more accurate experimental results. However, high-dimensional features can lead to the classification challenges in which data is measured and tracked at a very short sampling interval along with increment of time overhead and hardware demand in the precise classification. We

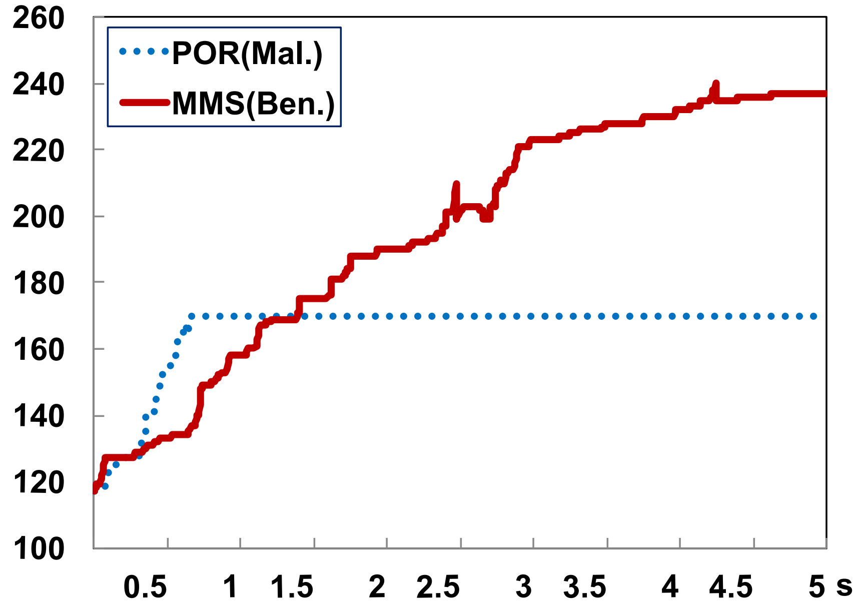

Figure 4.14: nr_ptes

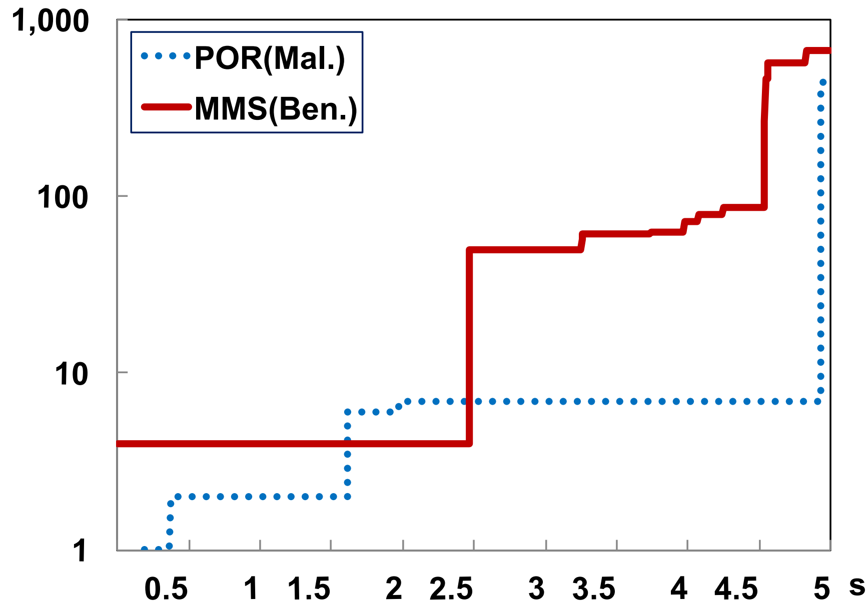

Figure 4.15: signal_nvcsw

designed dimensionality reduction models to percolate features and summarized the major and commonly-used features in Table 4.1 according to the weights of training examples.

We examined the features in Table 4.1 and their variation tendency between the fine time slices in Fig. 4.5, Fig. 4.6, Fig. 4.7, Fig. 4.8, Fig. 4.9, Fig 4.10, Fig. 4.11, Fig 4.12, Fig. 4.13, Fig. 4.14, Fig. 4.15, and Fig. 4.16. The benign application which is a safe and

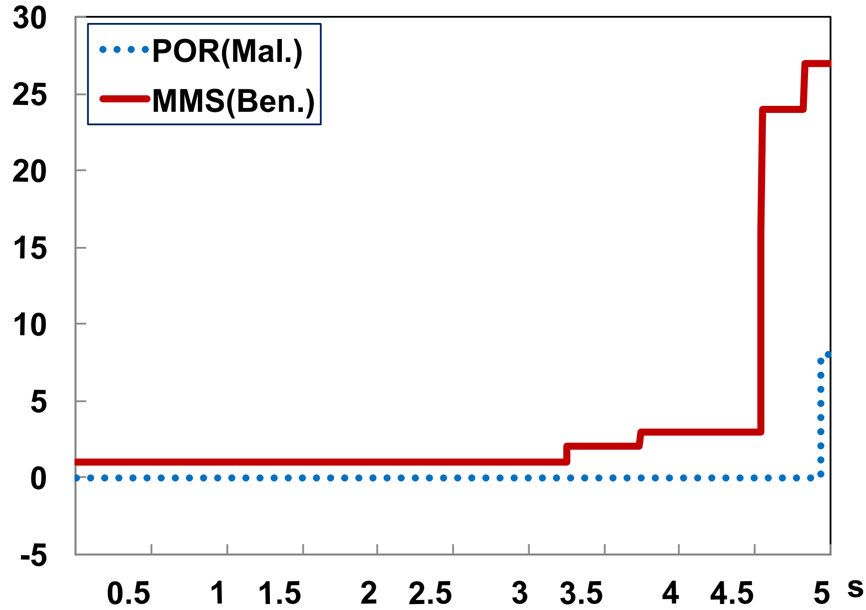

Figure 4.16: stime

stable finance APK (Android Package Kit) app implemented by Microsoft Corporation, MSN Money-Stock Quotes & News Free [1], is installed into the Android platform for analyzing feature trend. A Trojan virus, Porns.apk [2], accesses the information of telephone services, manipulates the SMS operation, and steals the account privacy. The interesting traits of the two APK apps are depicted in Fig. 4.5, Fig. 4.6, Fig. 4.7, Fig. 4.8, Fig. 4.9, Fig 4.10, Fig. 4.11, Fig 4.12, Fig. 4.13, Fig. 4.14, Fig. 4.15, and Fig. 4.16. As we can see from the 12 figures (a)-(l), none of the features of process execution converge at some steady state. In contrast, the malware lines of the selected features approach a smooth and steady status after a sharp growth. Take Fig. 4.8 shared_vm as an example, the vertical value of the red line representing benign application exceeds the blue dash line of malware applications at the 2nd second. The vertical unit of shared_vm parameter is 10^5 and the same as shown in Fig. 4.1. The points behind the crossing point (1.7, 3.57) fall into two clusters instead of aggregating to some points again as the value of shared_vm, but there are few noise points at

[1]http://apkpc.com/apps/com.microsoft.amp.apps.bingfinance.apk
[2]http://www.getjar.com && https://www.virustotal.com

point (2.75, 3.58) which disturbs the feature extraction after benign and malware processes become stable.

Unlike non-malware apps which request reasonable resources discretionarily, malware apps require resources at the beginning, thereafter maintaining a steady status over a long time duration. The parameters listed in Table 4.1 exhibit this distinct trends of Fig. 4.5, Fig. 4.6, Fig. 4.7, and Fig. 4.8, Fig. 4.9, Fig 4.10, Fig. 4.11, Fig 4.12, Fig. 4.13, Fig. 4.14, Fig. 4.15, Fig. 4.16 between malware and nonmalware. In detail, Fig. 4.5, 4.6, 4.7, and 4.8 indicate the number of different types of memory pages is related to malware detection. Besides, mapping counts of Fig. 4.9, the maximum of RSS from Fig. 4.10, major page faults in Fig. 4.13, and page table numbers in Fig. 4.14 are also significant to identify malicious behaviors. The signal information of volunteer context switches of Fig. 4.15, and system time from Fig. 4.16 have the similar trends with others. As shown as Fig. 4.11 and Fig. 4.12, it seems that the two parameters, the number of in-volunteer context switches (nivcsw) and the number of volunteer context switches (nvcsw), follow the linear distribution as the time goes on.

4.2 Data Processing

4.2.1 Data Cleaning and Data Filling

Each instance of the Android kernel structure includes 112 fields and each field value is translated into numerical format during data storage procedure. Detecting or correcting corrupt data instances is necessary for linear and nonlinear classifiers. Therefore, when half of the 112 fields of the data instance is inaccurate, we remove this record from its database. However, if only few fields are missed or corrupted, we replace these fields with their expected values to preserve data consistence.

Since the data collector analyzes Android applications for a short interval and obtains 15,000 data instances per .apk application while executing an application, there are repeated and redundant data instances for corrections of randomly missing data. After the data

cleaning is accomplished for repairing the data inconsistency, we remove the repeated data to improve the data quality by comparing each row of datasets with the rest of rows.

In order to approach ideal data for training classification models, standard procedures are required to improve the quality of data for modeling. After data cleaning and reducing, all data is converted into numerical value with a [0,1] or [-1,1] range by a normalization or standardization function for further model training. There are two methods to process these numerical data samples as shown as below, 0-1 scaling of Eq. (4.1) and Z-score scaling of Eq. (4.2),

$$X_i^{'} = (X_i - X_{min})/(X_{max} - X_{min}), X^{'} \in [0, 1] \tag{4.1}$$

$$X_i^{''} = (X_i - \mu_s)/\sigma_s \tag{4.2}$$

In Eq. (4.1), $X_i^{'}, X_i, X_{min}, X_{max}$ represent the new data value after transformation, the original data value in a column, the minimal value among all the data samples in the same column with each original value, the maximal value among all the data samples in the same column with each original value, respectively. $X_i^{''}, X_i, \mu_s, \sigma_s$ of Eq. (4.2) represent each new data value, each original value of a column, the expected value in the same column with each original value, and the standard deviation value in the same column with each original value.

The method in Eq. (4.1) is applied to normalize our streaming data generated by using the smartphones. Moreover, this method guarantees that all new features will be not more than one and not less than zero. A. Kusiak proved that the classification accuracy generated from datasets can be improved with this specific feature bundles [54].

4.2.2 Dimensional Reduction Methods

Traditional analytical procedure fails to further authenticate malware and non-malware data examples with such large dimensions. In mathematical domains, many methodologies decompose high dimensions to low dimensions, preserving the consistency of the original

data. The favorable techniques are mostly used in computer science for constructing predictive models. Linear and nonlinear dimension reduction techniques are very popular in machine learning when facing pattern recognition with such high-dimensional features. In this study, we extract some important features and assign the related weights in both methods.

Simplifying the issue of dimension reduction, we introduce the statement [40] on the problem as the following form: given a k-dimensional variable vector $\mathbf{x}$=$(x_1, x_2, ..., x_k)^T$, find a low-dimensional alternative, $\mathbf{y}$=$(y_1, y_2, ..., y_s)^T$ with $s \leq k$. Each variable of column vector $\mathbf{x}$ represents a feature in our dataset, where dimension $\mathbf{k}$ equals to the number of original parameters in Table 5 [1]. The vector $\mathbf{y}$ contains fewer or equal number of original features with $\mathbf{s}$ dimensions.

Assuming there are n data instances in total, each row is a k-dimension variable $\mathbf{x}$ with mean $E(\mathbf{x}) = \mu = (\mu_1, \mu_2, ..., \mu_k)$ and covariance matrix $E(\mathbf{x} - \mu)^T(\mathbf{x} - \mu) \triangleq \Sigma_{k \times k}$. We denote the training data matrix with n instances by $\mathbf{X}$= $x_{i,j}, 1 \leq i \leq k, 1 \leq j \leq n$, moreover μ_i as the mean of the i-th variable (feature) and σ_i as the i-th variable's standard deviation, respectively. We normalize the entire training matrix by $(x_{i,j} - \mu_j)/\sigma_j$, where each value of $\mathbf{X}$ subtracts the mean of the i-th variable with $\mu_i = 1/n \sum_{j=1}^{n} x_{i,j}$, divide the standard deviation $\sigma_i = 1/n \sum_{j=1}^{n} (x_{i,j} - u_j)^2$.

The linear dimension reduction yields the result of each $s \leq k$ component $\mathbf{y}$, being a linear weighted combination with original variables: $y_i = w_{i,1}x_1 + w_{i,2}x_2 + ... + w_{i,k}x_k$ with $i = 1, 2, ..., s$. This formula is transformed to describe all the reduced s dimensions as the below expression:

$$\mathbf{y} = \mathbf{W}\mathbf{x}, \tag{4.3}$$

where $\mathbf{W}$ is a $s \times k$ weight matrix, so the $\mathbf{k}$ dimensional vector $\mathbf{x}$ is mapped to a $\mathbf{s}$ dimensional vector $\mathbf{y}$ for $\mathbf{s} \leq \mathbf{k}$, the new combination $\mathbf{y}$ is the hidden features. There are many state-of-the-art and matured dimension reduction methods in machine learning literatures. In this

study, we investigate these strategies of principle component analysis, chi-squared statistic and information gain for lowering the dimension of the input data.

Principal Component Analysis (PCA) [48] attempts to find a linear combination of the original data with its variance in which a larger value represents the more important to dimension reduction. We have **k** dimensions in the training data and testing data and PCA generates **k** Principal Components (PCs) of the original data where not all PCs are necessary to reduce dimensions. The most important **s** variables are determined during the procedure of spectral decomposition of the covariance matrix Σ. Matrix Σ is a $k \times k$ square matrix, decomposing to be written as

$$\Sigma = \mathbf{U}^T \Lambda \mathbf{U}, \tag{4.4}$$

where $\Lambda = diag\left\{\lambda_1, \lambda_2, ..., \lambda_k\right\}$ is the diagonal matrix of singular values with the descending order $\lambda_1 \leq \lambda_2 \leq, ..., \leq \lambda_k$ and $\mathbf{U}$ is a $k \times k$ orthogonal matrix with singular vectors. The PCs are given by the **s** rows of $s \times n$ matrix $\mathbf{S}$ as below,

$$\mathbf{S} = \mathbf{U}\mathbf{X}, \tag{4.5}$$

where the first **s** singular vectors of matrix **U** contains the smallest standard deviation of **X**. Other linear dimension reduction techniques are very similar to PCA, decomposing the covariance matrix to the form as (4.4). Factor Analysis (FA) [22] proposes some assumptions of **s**-factor model where the k-dimension vector $\mathbf{x} = \mathbf{A}_{k \times s}\mathbf{f}_{s \times 1} + \mathbf{u}_{k \times 1}$ becomes the zero-mean one and **A**, **f**, **u** are constant matrix, common factors and specific factors, respectively. Additionally, the covariance matrix Σ is written as

$$\Sigma = \mathbf{A}\mathbf{A}^T + \mathbf{diag}(\varphi_{11}, \varphi_{22}, ..., \varphi_{kk}), \tag{4.6}$$

where $\mathbf{diag}(\varphi_{11}, \varphi_{22}, ..., \varphi_{kk})$ is the covariance matrix of μ which has $\mathbf{E}(\mu) = 0$, $\mathbf{Cov}(\mu_i, \mu_j)$ $= 0$ for $i \neq j$. This technique has several assumptions in decomposing covariance matrix,

while PCA seeks to the relevant singular values by decomposing its covariance matrix Σ without considering the mathematical hypothesis.

Chi-squared statistical dimension reduction [83] has been used in the high-dimensional data preprocessing as feature selection because it conserves the original data input without rotating the selected features. In addition, chi-squared technique proves less time-consuming than PCA in scientific computation. Compared to PCA, chi-squared method does not require to decompose a large covariance matrix, but rather calculate the values of independence of each variable. The chi-squared measure [56] evaluates the independence between the feature and categories. It constructs the contingency table according to the value of input feature as shown as Table 4.2, in which n_i is the number of instances whose feature value equals to c_i, and n_{mi}, n_{bi} are the frequency of the malware and benign instances with feature value c_i. There is the summation (n) of the number of all malware instances (n_m) and the number of all benign instances (n_b). The chi-squared metric of the i-th feature is defined in the following formula (4.7)

$$\chi_i^2 = \sum_{j=1}^{r} (\frac{n_{mj} - \mu_{mj}}{\mu_{mj}})^2 + (\frac{n_{bj} - \mu_{bj}}{\mu_{bj}})^2, \tag{4.7}$$

where $\mu_{mj} = n_j n_m / n, \mu_{bj} = n_j n_b / n$ are the expected values of the malware frequency and benign frequency with the input value c_j, respectively. The parameter r stands for the number of the instances c_j. The feature is more relevant to the accurate classification if the result from (4.7) is much larger than other features. We select s features empirically in all sorted features.

Correlation method [91] can identify the relevant features for high-dimensional data. For this approach, correlation coefficient is the most well-known measure to evaluate the goodness of selected features for classification. Suppose we have a class vector X and an

attribute vector Y, the correlation coefficient c is given by the following formula (4.8)

$$c = \frac{\sum(x_i - \bar{x_i})(y_i - \bar{y_i})}{\sqrt{\sum(x_i - \bar{x_i})^2}\sqrt{\sum(y_i - \bar{y_i})^2}}, \tag{4.8}$$

where $\bar{x_i}$ is the mean of X and $\bar{y_i}$ is the mean of Y. The higher value of c represents the more predictive feature to the class concept.

Information Gain [63] which is based on the information theoretical concept of entropy, can evaluate the association between features and categories as well as correlation technique. The entropy of a class vector X is defined as Eq. (4.9)

$$H(X) = -\sum_i P(x_i) log_2 P(x_i), \tag{4.9}$$

and the conditional entropy of X after an attribute variable Y is defined as Eq. (4.10)

$$H(X|Y) = -\sum_j P(y_j) \sum_i P(x_i|y_j) log_2 P(x_i|y_j), \tag{4.10}$$

where $P(x_i)$ is the probability of values of the variable X, $P(x_i|y_j)$ is the probability of the variable X given the values of Y, and $P(y_j)$ is the probability of Y. The information gain (IG) is given by the formula (4.11)

$$IG(X|Y) = H(X) - H(X|Y). \tag{4.11}$$

For the sake of dimension reduction, other linear and nonlinear techniques gain similar results of feature selection as well, such as information gain, correlation, gini index and SVM. Although these techniques contain a bit of differences in mathematical theory and implementation, the order of their weighted features exists similarity.

Table 4.2: Contingency Table of i-th Feature and Category in Training Set $\mathbf{X}$

Value	M(Malware)	B(Benign)	SUM
$c_1 = x_{i1}$	n_{m1}	n_{b1}	$n_1 = n_{m1} + n_{b1}$
$c_2 = x_{ij}$	n_{mj}	n_{bj}	$n_2 = n_{mj} + n_{bj}$
$c_3 = x_{ik}$	n_{mk}	n_{bk}	$n_3 = n_{mk} + n_{bk}$
...	...	...	...
$c_r = x_{il}$	n_{ml}	n_{bl}	$n_r = n_{ml} + n_{bl}$
$j, k, l \in (1, n]$	n_m	n_b	$n = n_m + n_b = \sum_{s=1}^{r} n_s$

4.3 Local Machine Learning Methods

Naive Bayes This probabilistic model [66] trains the dataset based on Bayes' theorem considering the strong independence between each features. The Naive Bayes classifiers are highly scalable and linear in the number of features of a learning problem, which requires a number of parameters. Maximum-likelihood training can be done by evaluating a closed-form expression, which takes linear time, in stead of expensive iterative approximation as other types of classifiers. This statistical model requires to achieve the expected values and variances from a large amount of input data while training a suitable detection model.

Decision Tree Decision tree classifier [52] is one of the predictive modelling approaches for data mining and machine learning. This classifier models can take a finite set of values. In these tree structures, leaves represent class labels and branches represent conjunctions of features that lead to those class labels. The classifier of decision tree adopts recursive partition to find the best tree from a given dataset. Thus it utilizes the divide and conquer method to perform the training of the input data. In contrast to other methods, decision tree is straightforward to interpret in a clear graph with nodes (denotes the relevant feature), leaves (denotes the categories), and edges (denotes the range of node values).

Neural Network Neural Network applies the mathematical model [45] to learn the characteristics of the input dataset. It can construct neurons of first layers according to the input features, then decide the weights of each feature in a feed-forward neural network by back-propagation. The second layer contains two neurons as the output, generally, the number

of the hidden layers does not exceed four. More complicated neural network will have more layers of neurons, some have increased layers of input neurons and output neurons. The synapses parameters are called weights that manipulate the data during the calculation of the whole network.

4.4 Parallel Malware Detection

4.4.1 In-Memory Classification

Resilient Distributed Datasets (RDDs) of a Spark system are used to perform in-memory computations and stored in shared memory for machine learning's iterations [93]. RDD programming can be implemented in Scala [8], which is an object-oriented, efficient programming language. The Spark system loads gigabytes of datasets to a HDFS file [24] from a local node and transforms them into a RDD for the further data processing. There are fundamental RDD operations [92], such as *map(f: T $\Rightarrow$ U), filter(f: T $\Rightarrow$ Bool), reduceByKey(f: (K,V) $\Rightarrow$ V), etc.*, to promote iterative machine learning algorithms.

To expedite the computation for a large dataset, we introduce the in-memory classification algorithm on Spark systems. As shown in Algorithm 1, Mesos cluster [47] and Spark master [93] are configured by Spark configuration parameters and functions (Line 3 and Line 4). $conf$ and *sc* denote the configuration parameters. Then the original data is processed into the RDDs format (*Data*) stored in memory and is mapped to (K, V) pairs for further predictions (Lines 7-8). RDDs are cleaned and transformed for precise classifications when the flags $CleanData$ and $TransformData$ are set by customers (Lines 9-13). When training the predicting models, the detailed parameters, Pre_MSE(Mean Squared Error), Pre_Iter (# of Iterations) are specified by users. Through training the data iteratively, finally the customer can obtain a model stored in memory until the status reaches the convergence (Lines 15-19). Predicting the new data samples by applying the in-memory model will be executed in Line 21 and then the predicted results will be returned to the user.

Algorithm 1 In-Memory Classification on Spark

```
1: Initialization
2: // Set Configuration of Mesos and Spark Masters
3: conf ← SparkConf().setMaster();
4: sc ← SparkContext(conf);
5: Data Processing
6: // Parse Input of .CSV files to fields and map data to (K, V) pairs in memory
7: Data ← sc.textFile().split();
8: Data.map(r ⇒ (Labels, Vectors));
9: if (CleanData ∧ TransformData) then
10:    CleanAndTransform(Data);
11: else if (CleanData) then
12:    Clean(Data);
13: end if
14: Train Models and Predict Results
15: while (MSE < Pre_MSE) ∨ (Iter < Pre_Iter) do
16:    Model ← TrainModelWithAlgo(Data);
17:    Outputs ← ApplyModel(Data);
18:    MSE ← SUM((Outputs − Labels)^2);
19: end while
20: while (PredictData) do
21:    Output ← ApplyModel(PredictData);
22:    Return Output;
23: end while
```

4.4.2 Parallel Classifiers

In our design, four popular classifiers are used to detect the Android malware: Decision Tree, Naive Bayes, Logistic Regression and Support Vector Machine. Here we denote Decision Tree, Naive Bayes, Logistic Regression and Support Vector Machine as DT, NB, LR, and SVM. In addition, we compare the four algorithms based on the ROC (Receiver Operating Characteristic) curve which is a metric for binary classifiers [25] and execution time with different computing nodes. Fig. 4.17 shows the ROC space of four classifiers, DT, NB, LR, and SVM. When applying these classifiers to our datasets, they generate four separate confusion matrices that in turn correspond to ROC points [39]. The X-axis denotes the false positive rate which equals to # of negatives incorrectly classified divides by # of total negatives. The Y-axis denotes the true positive rate which equals to # of positives

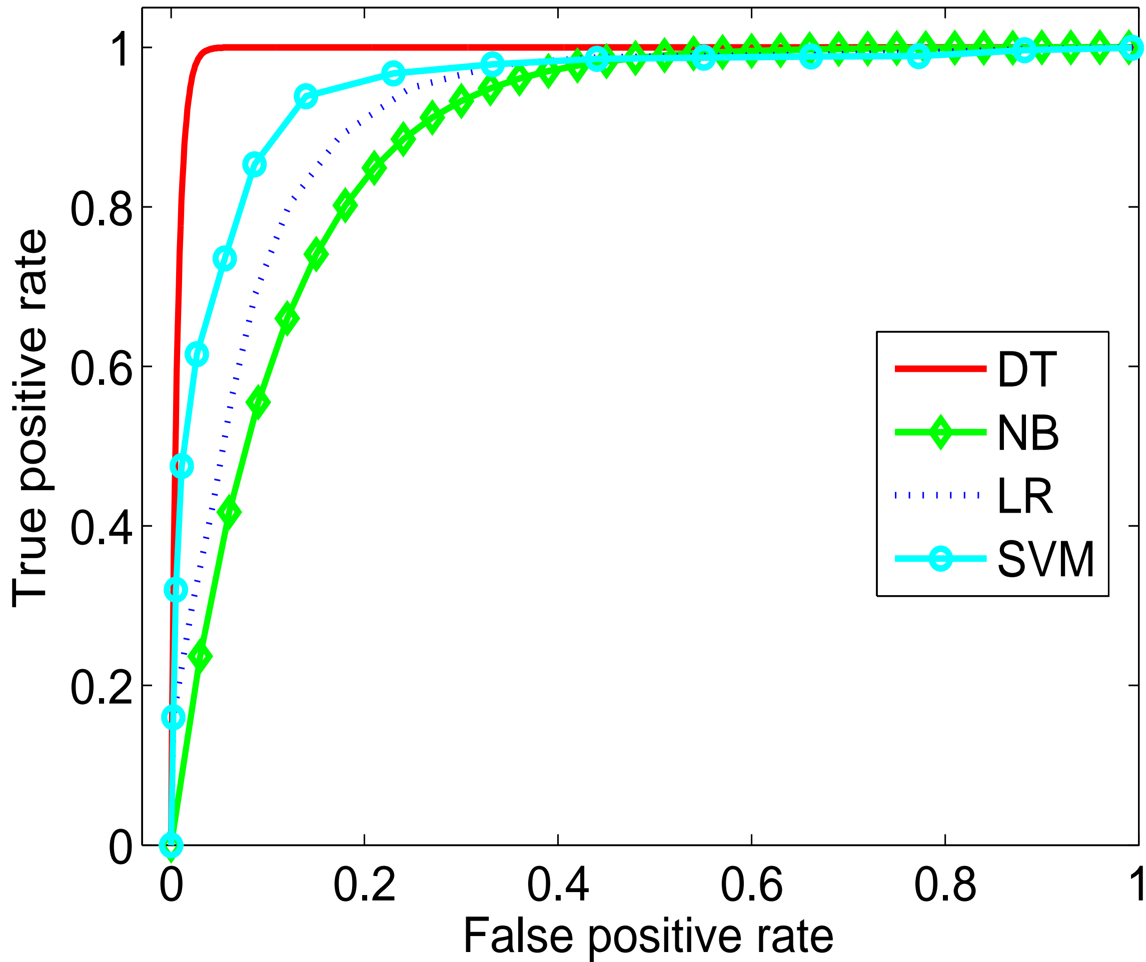

Figure 4.17: ROC Curve of four classifiers

correctly classified divides by # of total positives. These results indicate our kernel features from Android can be used to detect malware with a high accuracy. As we can see, DT curve as the best classifiers among the four classifiers raises rapidly and sharply to the maximum y-axis value. In contrast to DT, NB curve as a less accurate classifier increases slowly and smoothly. Area Under an ROC Curve (AUC) [25] is an easy way to compare classifier performance with a scalar value or a graphics demonstration. From Fig. 4.17, DT classifier has the largest AUC area, the next are SVM and LR classifiers, and NB classifier has the smallest AUC area (a larger AUC value represents a better and more accurate classifier).

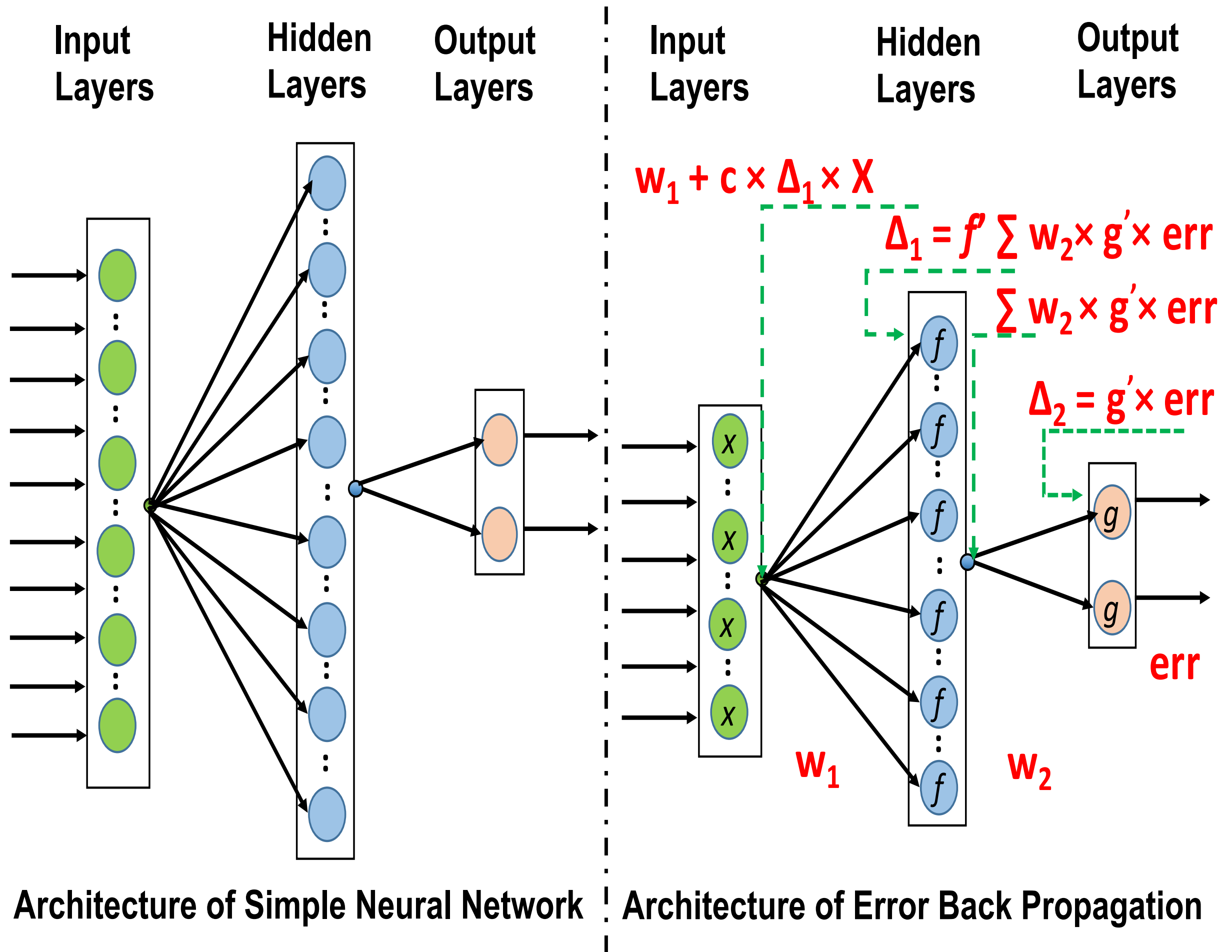

Figure 4.18: Architecture of Error Back Propagation Algorithm

4.5 Designs of Neural Network

4.5.1 Traditional Neural Network (EBP)

In the EBP network, there are similar layers to the simple ANN architecture. However, EBP requires to return errors between the outputs and the targets of the previous step of numerical calculation. In Fig. 4.18, the errors are propagated back to all the former layers. The error vector, err, is the difference between the known target value $target \rightarrow t$ and the calculated value $output \rightarrow o$ given by the activation function 4.12

$$o_j = f(net_j) = \frac{1}{1+e^{-\beta net_j}} \tag{4.12}$$

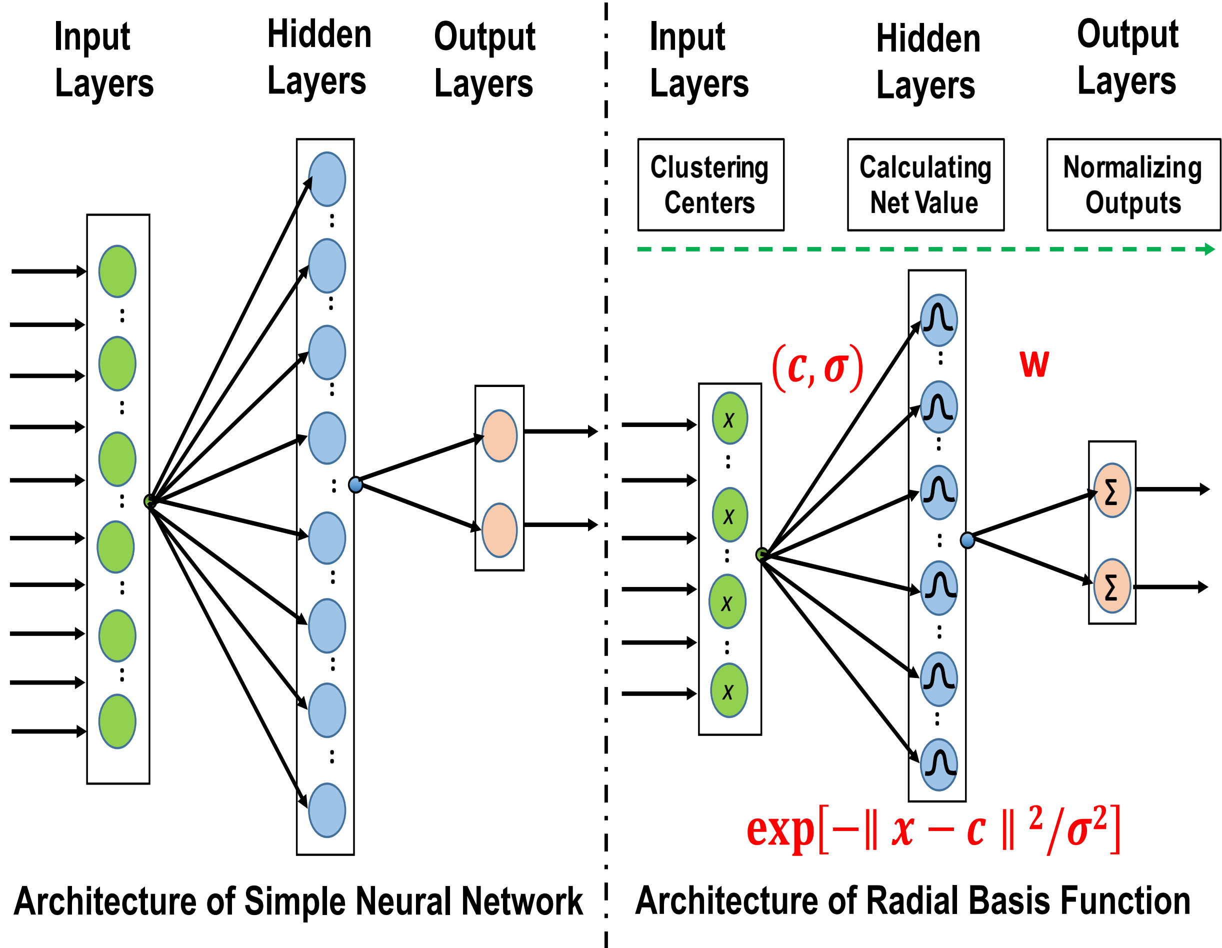

Figure 4.19: Architecture of Radial Basis Function

where β is the constant of the activation function. A smaller value of β leads to a soft transition and a larger value of β can cause a hard transition in activation levels. In our experiments, we set β to 1 by default.

EBP algorithm [67] provides a canonical derivation using the squared errors as in equation 4.13:

$$err = \sum_{j=1}^{n}(t_j - o_j)^2 = \sum_{j=1}^{n}(t_j - f(net_j))^2 \tag{4.13}$$

where j is the number of patterns from 1 to n, t_j, o_j are the target value and the output value of the j-th pattern and function f is the same function as in equation 4.12. To reduce the error rate of the feedforward computation, ANN needs to backward pass the error signal for each neuron and update the weights of each neuron in hidden layers. The weight updating

rule is demonstrated as the following equation:

$$W_{k+1} = W_k + c \times \Delta \times Input \tag{4.14}$$

where W_{k+1}, W_k represent the new weight vector and the former weight vector in the same hidden layers, respectively. Parameter c is the learning constant to control the step size of correcting the errors and Δ denotes the inner product of the errors of the present layer and the derivatives of the output function. $Input$ is the input value of each layer, note that middle hidden layers receive their input value from the previous layer unlike the first hidden layer which directly obtains its input value from the original or reduced input without the calculation of intermediate ANN layers.

Fig. 4.18 indicates the procedure of updating weights for neurons in the same hidden layer as in equation 4.14. The value of Δ_2 in the output layer is computed by multiplying err to the derivative of the output function g'. In hidden layers, the errors have to be calculated by summating the product of weight W_2 and Δ_2. Similarly, the value of Δ_1 of the hidden layers is given by equation 4.15:

$$\Delta_1 = f' \times \sum W_2 \times \Delta_2 = f' \times \sum W_2 \times g' \times err \tag{4.15}$$

Following the weight updating rules of 4.14, the weights of the hidden layers, W_1, can be updated by adding the product of learning rate c, Δ_1 and the input value X, which are also shown in Fig. 4.18. With thousands of iterations of updating weights, the overall error rate can be reduced to a lower level.

4.5.2 Enhanced Neural Network

Radial Basis Function (RBF) [29] network is proposed to solve the problems of nonlinear classifications or nonlinear approximation. A typical EBP network attempts to reduce the global error rate through less iterative calculations. However, EBP network is not suitable

for the exascale computation of millions of data samples. RBF network utilizes a Gaussian kernel in the hidden layers to accomplish the nonlinear transformation of input samples, which enhances the training performance of exascale datasets.

Fig. 4.19 shows the architecture of a RBF network, where the RBF network contains three similar layers to EBP, input layers, hidden layers and output layers. Traditionally, its input layers provide the original or reduced datasets for its hidden layers. However, its hidden layers perform nonlinear transformation and maps the original space to a new space using the following equation 4.16:

$$net_j = exp(-\left\|X - C_j\right\|^2 / \sigma_j^2) \tag{4.16}$$

where net_j, X, C_j, σ represent the j-th neuron's net value, the input vector, the j-th neuron's center position, and the j-th neuron's standard deviation, respectively. $\|.\|$ denotes the Euclidean norm and $\|X - C_j\|$ stands for the Euclidean distance between the pattern and the center. The RBF network utilizes the center's information C and its standard deviation between input layers and hidden layers.

The output layers of a RBF network need to combine each output from hidden layers as in given function 4.17:

$$o = \sum_{j=1}^{np} W \times net_j \tag{4.17}$$

where o, j, W, net_j represent the output value of output layers, the number of neurons from 1 to np, and the j-th neuron's net value, respectively. Actually, this step of RBF network demonstrates how to select the closest center (normalizing the output) for the input value among all the net values by 4.16. RBF network includes three steps: clustering centers, calculating net values and normalizing outputs. The step of clustering centers is demanded to be finished by the clustering algorithms before applying the RBF transformation. Then, the transformation given by equation 4.16 can calculate the net values, meanwhile output values also can be calculated according to the output function 4.17.

4.5.3 RBF Network Design and Implementation

In this section, we explain how to design a practical RBF network and implement it for effectively classifying Android malware and goodware. From the brief description of the RBF network in Section 2.3, a RBF network needs retrieve the centers of data points' before its nonlinear transformation. Therefore, we choose the K-means algorithm to calculate the data points' centers. Then, according to the data centers, we can train easily the RBF neurons for each data point and evaluate its accuracy rate.

K-means to Calculate RBF Centers

K-means algorithm can separate a clustering of data points into K regions. It selects the K centers randomly before its first iteration and then iteratively performs the following steps:

1. Calculate the centroid which is closest to the current data point and assign the current data point to this cluster.

2. Update the selected cluster which is the closest to the current data point with the mean of the new dataset including the current data point.

Algorithm 2 explains how K-means clustering methods are used to find a locally optimal partition of large datasets. Firstly, we initialize them by randomly choosing k data samples from the dataset or using the previous results (Lines 2-6). S_j is the sum of all the data points belonging to the j-th center, n_j is the total number of all the data points belonging to the j-th center. During the procedure of iterative computation of k clustering regions, additional variables, sum_j and n_j, are required to temporally save the intermediate results (Line 7). The K-means algorithm assigns each data point x_i to a region D_j that has the closest centroid to x_i, and calculates the relevant cluster statistics (Lines 9-16). Meanwhile, the centroids of the k clusters are updated with the mean of the dataset of these clusters (Lines 17-19). The execution of this algorithm can be terminated until all the centroids of the

Algorithm 2 K-means clustering

```
1: Input: Training dataset D, number of clusters k
2: if the first iteration then
3:     Initialize the k clusters randomly
4: else
5:     Read the k clusters c_j from the last step
6: end if
7: Set sum_j = 0 and n_j = 0 for j = {1, ..., k}
8: while TRUE do
9:     for x_i ∈ D do
10:        for j ∈ {1, ..., k} do
11:            j_min = arg min ||x_i − c_j||
12:            sum_{j_min} = sum_{j_min} + x_i
13:            n_{j_min} = n_{j_min} + 1
14:            D_j ← x_i
15:        end for
16:    end for
17:    for j ∈ {1, ..., k} do
18:        c_j = sum_j / n_j
19:    end for
20: end while
```

k clusters rarely changes or the number of iterations exceeds the threshold that customers set.

Select the Kernel Widths (σ)

From the activation function of the RBF network, we require the centroid c_j and the standard deviation σ_j to decide the curve of the Gaussian Function. Section 4.5.3 has introduced how to calculate the centroid c_j and we explain how to effectively select the kernel width σ_j in this section. A very larger or small σ_j, the kernel width [21], can cause the numerical issues with gradient descent algorithms. Therefore, we adjust the kernel widths dynamically based on the different parameters of the Gaussian basis function.

The kernel width can be determined by different setting schemes [72]. In this study, we investigate a popular method for the setting of the kernel widths. In these cases, the K-means method is utilized to calculate the centroid c_j. The kernel width σ_j is set to the

mean of Euclidean distances between data points and their cluster centroid as the following equation (4.18)

$$\sigma_j = \frac{1}{n_j} \sum_{x \in D_j} \|x - c_j\| = \frac{1}{n_j} \sum_{x \in D_j} (x_i - c_{ij})^2 \tag{4.18}$$

In this situation of kernel widths, the values of the parameters n_j, D_j, c_j, which represent the number of data points belonging to the j cluster, the collection of the j cluster, and the j clustering center, respectively, are retrieved from the Algorithm 2.

Gradient Descent to Reduce the Error Rate

The RBF network can iteratively reduce the error rate by gradient descent to obtain the minimal error (4.19)

$$TE = \sum_{i=1}^{n} \sum_{j=1}^{k} (t_{i,j} - o_{i,j})^2 \tag{4.19}$$

where $t_{i,j}$ is the target response of the i-th output from the j-th neurons and $o_{i,j}$ is the actual response of the i-th output from the j-th neuron. Actually, the value of $t_{i,j}$ is known and the value of $o_{i,j}$ is achieved by the equation (4.17). The minimal error is that the derivatives of the parameters clustering center c_j, kernel width σ_j and the output weight w_j vanish. Therefore, an iterative computation of the gradient descent with the direction of the negative gradient $-\frac{\partial TE}{\partial w}, -\frac{\partial TE}{\partial c}, -\frac{\partial TE}{\partial \sigma}$ can solve this issue.

Combining the Gaussian basis function with the error reduction of the gradient descent, the updating rules of the RBF network can be obtained as the following equations (4.20), (4.21), and (4.22):

$$\Delta w_j = -\alpha \sum_{i=1}^{n} net_j(x_i)(t_{i,j} - o_{i,j}) \tag{4.20}$$

$$\Delta c_j = -\alpha \sum_{i=1}^{n} net_j(x_i) \frac{x_i - c_j}{\sigma_j^2} \sum_{j=1}^{k} w_j (t_{i,j} - o_{i,j}) \tag{4.21}$$

$$\Delta \sigma_j = -\alpha \sum_{i=1}^{n} net_j(x_i) \frac{(x_i - c_j)^2}{\sigma_j^3} \sum_{j=1}^{k} w_j (t_{i,j} - o_{i,j}) \tag{4.22}$$

Algorithm 3 Gradient Descent with Constant Learning Rate

1: **Input:** Training dataset D, α, TE_{min}, clustering centers set C, kernel width set σ
2: Randomly choose the weights vector W, initialize the target output vector TP and the input vector X with dataset D
3: **while** $TE > TE_{min}$ **do**
4: $NET = EXP(-||X - C||^2/\sigma^2)$
5: $OP = W * NET$
6: $\Delta w = -\alpha * NET * (TP - OP)$
7: $\Delta c = -\alpha * NET * (X - C)/\sigma^2 * W * (TP - OP)$
8: $\Delta \sigma = -\alpha * NET * (X - C)/\sigma^3 * W * (TP - OP)$
9: $W = W + \Delta w, C = C + \Delta c, \sigma = \sigma + \Delta \sigma$
Compute the new total errors TE
10: $OP2 = W * NET$
11: $ERR = TP - OP2$
12: $TE = sum(sum(ERR. * ERR))$
13: **end while**

where α is the learning rate constant which is significant to control the convergence to a minimum [16]. Here we set the learning rate to a small constant value to simplify the training procedure and avoid overshooting the minimal errors. Algorithm 3 shows the procedure of the gradient descent with the constant learning rate. The input values of Line 1 are obtained from Algorithm 2. In the iterative computation, the three values, $\Delta w, \Delta c, \Delta \sigma$, are used to update the previous values of W, C, σ (Lines 6-9). Then the new total errors can be triggered (Lines 10-12).

4.6 Multiple Dimensional Kernel Feature Collector

The multiple dimensional kernel feature's collector shown in Fig. 4.20(a), running both on Android devices and storage servers, is mainly composed of three components: (1) The scheduling mechanism of a malware repository, (2) message (package) communication, and (3) data processing of compression in the Android kernel module, transformation and storage via several User Datagram Protocol (UDP) services of lightweight data package transmission and Hypertext Transfer Protocol (HTTP) of the request-response module. In particular, with our stated aim of automatically scanning the malicious information of the current task

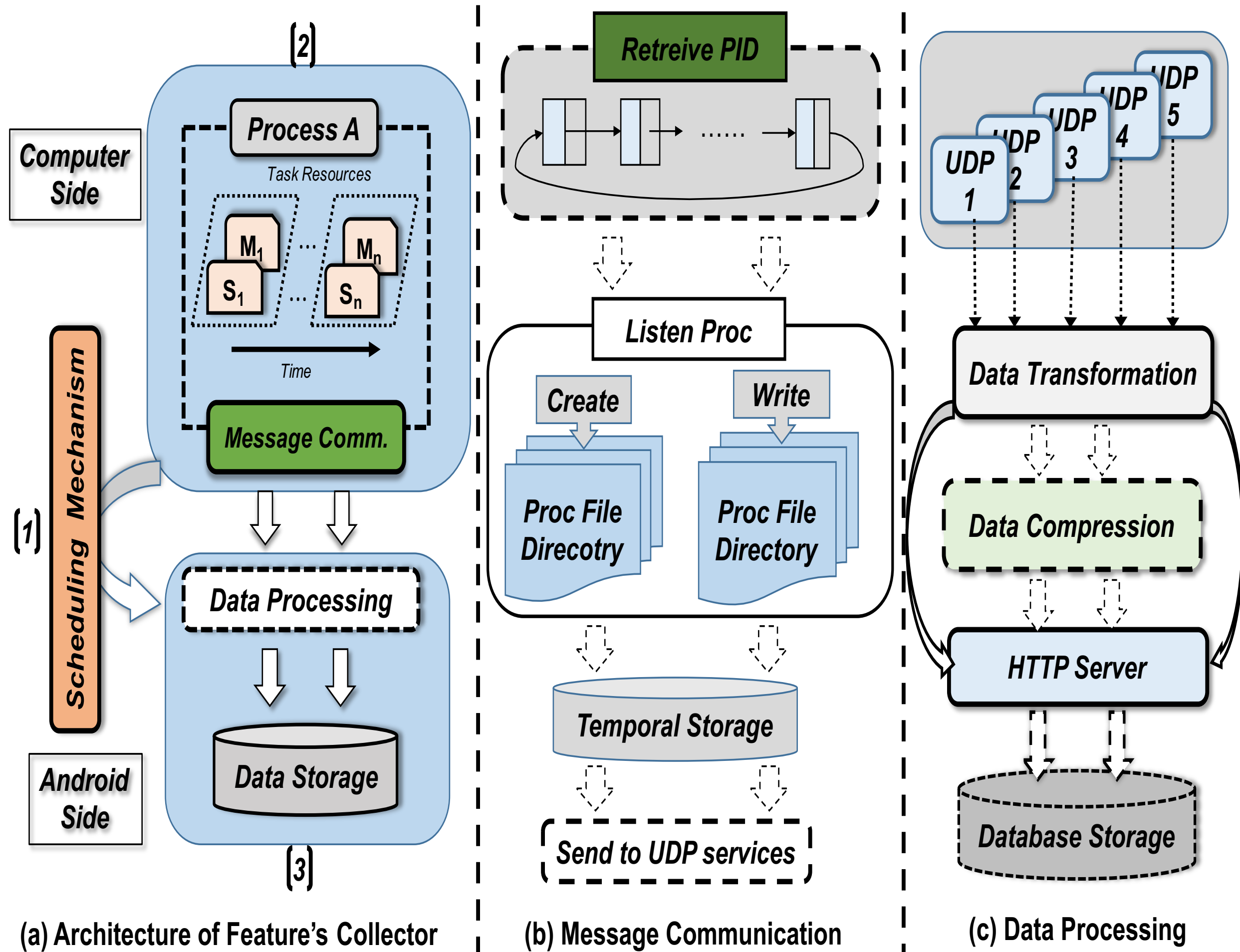

Figure 4.20: Overview of Multiple Dimensional Kernel Feature's (Raw Data) Collector. In (b), Message Communication Module in Local Computer . In (c), Data Processing Module in Android Kernel.

structure, the scheduling mechanism is designed and implemented to dispatch the setup of malicious apps concurrently. Note that in fact, customers do not demand to retrieve such large amount of data since they probably have installed a few malicious apps in their devices. Here for convenience of scanning a lot of Android apps, we utilize our lightweight scheduler based on First Come First Serve (FCFS) to manage their execution and scanning.

In our study, the scheduling component (1) in Fig. 4.20(a), is responsible for managing the task switching between malware repositories located in the hard disk and Android devices connected with the computer. The malware repository contains hundreds of malicious and benign APK files with the format of .apk. The scheduler running on the computer side

can issue APK files from the temporal queuing pool in the Android side. To ensure the atomicity of data records for all APK files, the process identifier of current programs as a unique attribute is utilized to differentiate repetitive applications.

Furthermore, the message communication shown in Fig. 4.20(b) creates the intermediate files with the format of proc file that is a hierarchical virtual filesystem and is able to read the information of all task structures (**task_struct**) from the Android kernel. While loading the module into Android devices, message communication assignments, such as memory allocation, file read operation, file write operation, are executed in coordination with the scheduling component. Likewise, monitoring the available data and reducing the repeated data are indispensable procedures to the system's maintenance and succinctness. In this part, a temporal storage pool saves the data package from running processes and transits these messages via a UDP connection.

Additionally, the data processing component in Fig. 4.20(c) is in charge of data format conversion, data compression and data transferring, which is divided into two parts: UDP and HTTP. UDP services offer the data format conversion from binary to string format for facilitating numerical calculation in future work. Meanwhile, some of the data is also compacted to another format with less bytes. When the conversion and compression are finished, this wrapped data pushes ahead by means of a HTTP server, triggering the data transference from the temporal storage pool to the local database.

4.7 Normalized Feature Weights

4.7.1 Distribution of Normalized Feature Weights

Fig. 4.21, Fig. 4.22, Fig. 4.23, and Fig. 4.24 demonstrate the normalized weights' distribution of 112 task parameters with PCA, Correlation, Chi-square and Info Gain methods and Table 5.3 shows the detailed distribution of these parameters. As we can see in Fig. 4.21, 4.22, 4.23, 4.24, and Table 5.3, 2.1 [1], **mem_info** and **signal_info** achieves the

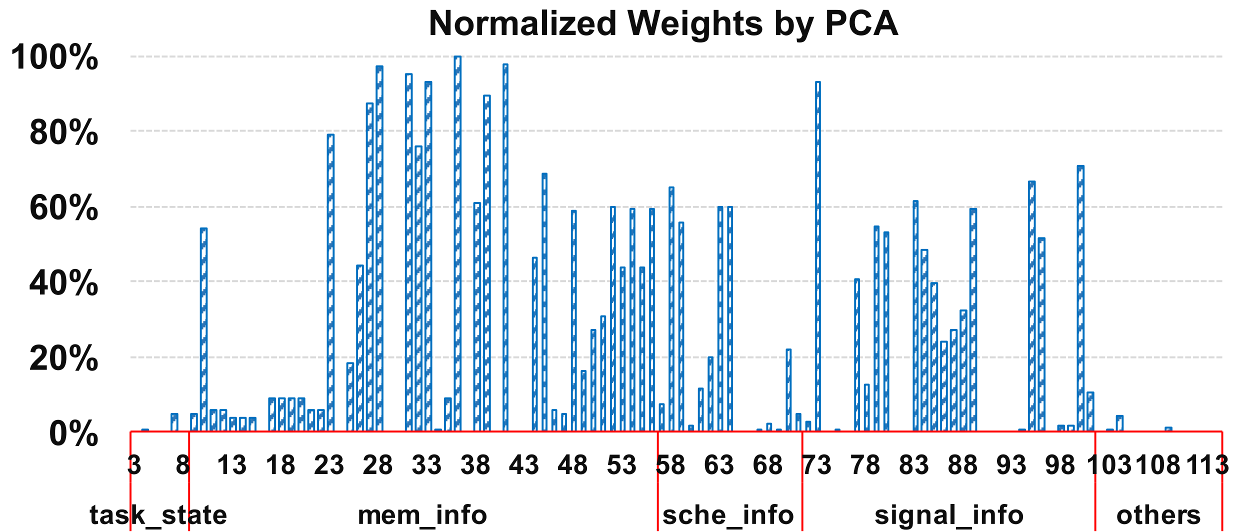

Figure 4.21: Normalized Weights Distribution of 112 Parameters with PCA method (**mem_info** & **signal_info** top 2 most popular)

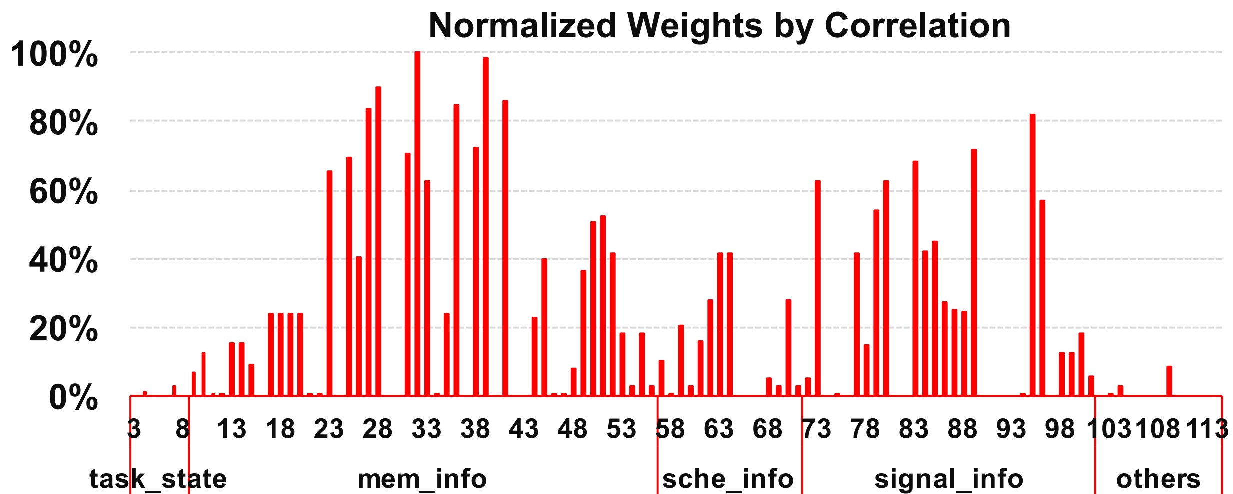

Figure 4.22: Normalized Weights Distribution of 112 Parameters with Correlation method (**mem_info** & **signal_info** top 2 most popular)

maximum number of normalized weights in (50%-100%). Theoretically, the 2 sets of task parameters precisely identify malware behaviors from similar behaviors.

In Fig. 4.21 of normalized weights distribution of 112 task parameters with PCA, x-axis value denotes the parameter's number and category in Table 5 [1] and y-axis denotes the

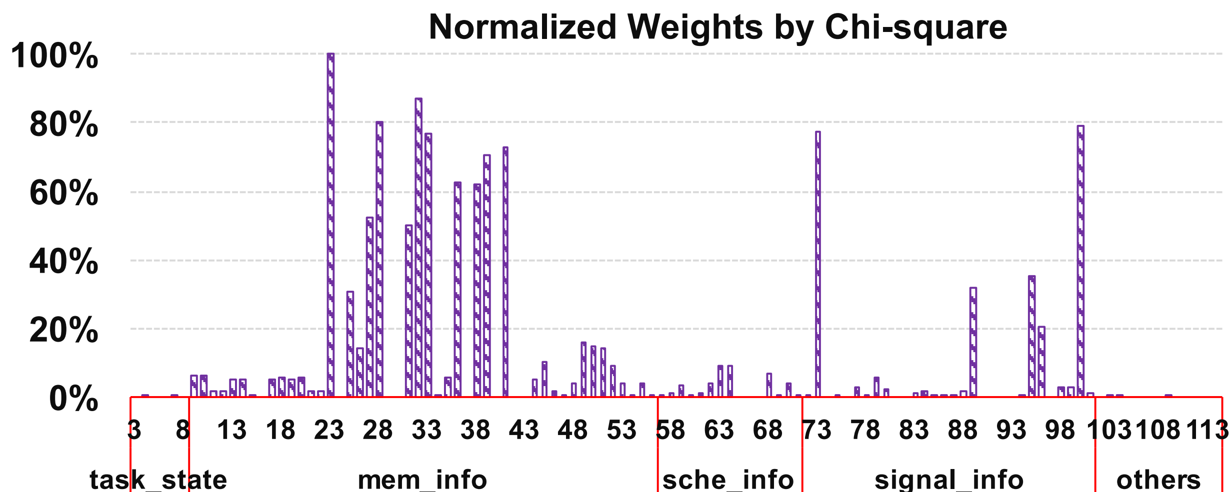

Figure 4.23: Normalized Weights Distribution of 112 Parameters with Chi-square method (**mem_info** & **signal_info** top 2 most popular)

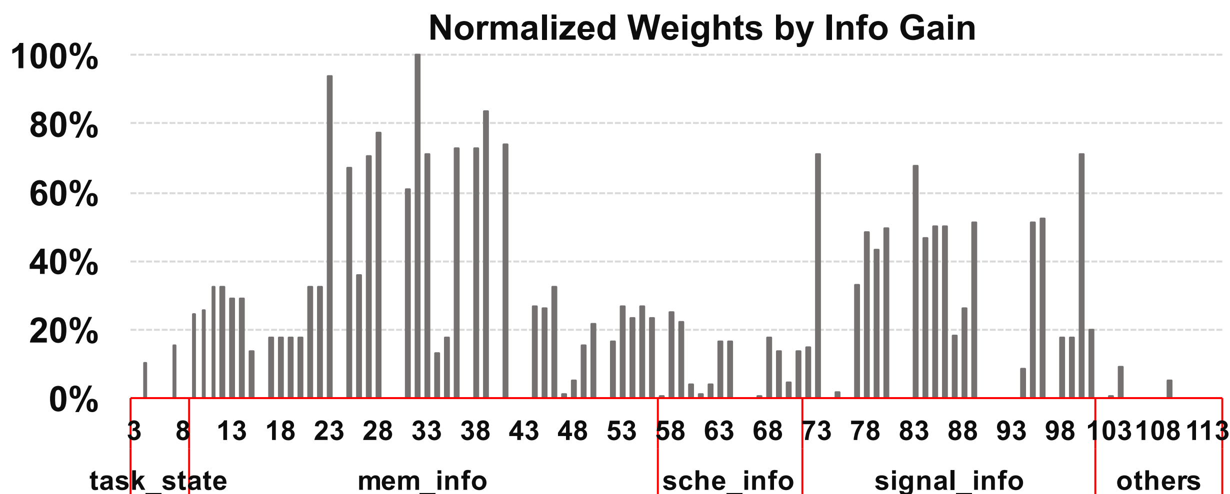

Figure 4.24: Normalized Weights Distribution of 112 Parameters with Info Gain method (**mem_info** & **signal_info** top 2 most popular)

value of normalized weights; **mem_info** reserves the maximum number of higher normalized weights (above 50%), **signal_info** also holds larger normalized weights than **sche_info** and **others**. First of all, **mem_info** features indicate the correlation of categorizing malware and nonmalware behaviors. Among 48 **mem_info** parameters, 16 out of 48 **mem_info** parameters retain the range (50% - 100%] of normalized weights , 8 parameters retain the

range [10% - 50%], 16 parameters retains the range (0% - 10%) of normalized weights, the 8 rest of **mem_info** do not achieve any weights. Secondly, **signal_info** exhibits more correlated to classification than **sche_info**, **task_state** and **others**. As we can see from the Fig. 4.21, in 28 **signal_info** parameters, 7 parameters achieve the range (50% - 100%] of normalized weights and 7 parameters spread in the range [10% - 50%], 5 of them achieves the weights in (0% - 10%), the 9 rest parameters have 0 weights. For 15 **sche_info** parameters, 4 parameters achieve the range (50% - 100%] of normalized weights, 3 out of them locate in the range [10% - 50%], 6 parameters only achieve (0% - 10%) of normalized weights, the 2 rest parameters are 0 weights. Furthermore, 6 **task_state** parameters and 14 **others** parameters achieve smaller weights 10%.

From Fig. 4.22, we can see the weights distributions of **mem_info** and **signal_info** are similar by 2 different dimensional reduction algorithms. 48 **mem_info** parameters are comprised of 13 parameters in (50% - 100%], 15 parameters in (10% - 50 %], 16 parameters in (0% - 10%] and 8 parameters equal to 0%. **signal_info** parameters contain more parameters with large weights than **sche_info**, **task_state** and **others**. 7 out of 28 **signal_info** parameters achieve larger normalized weights (above 50%), 9 parameters locate in the range [10% - 50%] of normalized weights, 3 parameters attain the weights in (0% - 10%), the 9 rest parameters have 0 weights. **sche_info** does not achieve larger normalized weights, which only includes 7 parameters in the range [10% - 50%], 5 parameters in (0% - 10%) and the 3 rest equal to 0. 6 **task_state** parameters and 14 **others** parameters achieve smaller weights 10% like PCA results.

In Fig. 4.23, x-axis value denotes the parameter's number and category in Table 2.1 [1] and y-axis denotes the value of normalized weights; **mem_info** and **signal_info** achieve larger normalized weights (above 50%) than **sche_info** and **others**, however, generally other normalized weights are less than 20%.

In Fig. 4.24, x-axis value denotes the parameter's number and category in Table 2.1 [1] and y-axis denotes the value of normalized weights; **mem_info** and **signal_info** achieve larger normalized weights (above 50%) than **sche_info** and **others**.

4.7.2 Details of Normalized Feature Weights

Table 4.3 shows the details of normalized feature weights for 112 kernel features. According to these weights, we can select features with normalized weights (from 0% to 100%, high weights are more than 40% and less than 100%) to train the classification models.

Table 4.3: Normalized Weights of 112 Task Parameters with PCA, Correlation, Chi-square and Info Gain

	#	PCA	Correlation	Chi-square	Info Gain	Sum
hash_key	1					
	2					
task_state	3	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
	4	2.32E-03	1.04E-02	4.58E-05	1.03E-01	1.16E-01
	5	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
	6	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
	7	5.07E-02	2.93E-02	6.66E-04	1.52E-01	2.33E-01
	8	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
mem_info	9	4.84E-02	6.84E-02	6.42E-02	2.43E-01	4.24E-01
	10	5.40E-01	1.23E-01	6.52E-02	2.58E-01	9.86E-01
	11	5.65E-02	3.22E-03	1.44E-02	3.25E-01	3.99E-01
	12	5.65E-02	3.22E-03	1.44E-02	3.25E-01	3.99E-01
	13	3.94E-02	1.55E-01	5.10E-02	2.89E-01	5.35E-01
	14	3.94E-02	1.55E-01	5.10E-02	2.89E-01	5.35E-01
	15	3.55E-02	9.10E-02	2.36E-03	1.37E-01	2.65E-01
	16	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
	17	8.76E-02	2.39E-01	5.17E-02	1.75E-01	5.53E-01
	18	8.76E-02	2.39E-01	5.60E-02	1.75E-01	5.58E-01
	19	8.76E-02	2.39E-01	5.17E-02	1.75E-01	5.53E-01
	20	8.76E-02	2.39E-01	5.60E-02	1.75E-01	5.58E-01
	21	5.65E-02	3.22E-03	1.44E-02	3.25E-01	3.99E-01

Continued on next page

Continued from previous page

#	PCA	Correlation	Chi-square	Info Gain	Sum
22	5.65E-02	3.22E-03	1.44E-02	3.25E-01	3.99E-01
23	7.93E-01	6.55E-01	1.00E+00	9.40E-01	3.39E+00
24	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
25	1.81E-01	6.93E-01	3.10E-01	6.70E-01	1.85E+00
26	4.43E-01	4.03E-01	1.43E-01	3.60E-01	1.35E+00
27	8.74E-01	8.34E-01	5.23E-01	7.04E-01	2.93E+00
28	9.70E-01	8.97E-01	8.02E-01	7.75E-01	3.44E+00
29	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
30	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
31	9.50E-01	7.05E-01	4.99E-01	6.09E-01	2.76E+00
32	7.59E-01	1.00E+00	8.70E-01	1.00E+00	3.63E+00
33	9.32E-01	6.24E-01	7.68E-01	7.11E-01	3.04E+00
34	1.32E-03	1.39E-03	1.50E-06	1.33E-01	1.36E-01
35	8.76E-02	2.39E-01	5.60E-02	1.75E-01	5.58E-01
36	1.00E+00	8.46E-01	6.24E-01	7.30E-01	3.20E+00
37	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
38	6.07E-01	7.25E-01	6.23E-01	7.29E-01	2.68E+00
39	8.92E-01	9.86E-01	7.07E-01	8.39E-01	3.42E+00
40	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
41	9.75E-01	8.57E-01	7.30E-01	7.38E-01	3.30E+00
42	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
43	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
44	4.64E-01	2.28E-01	5.10E-02	2.66E-01	1.01E+00
45	6.86E-01	4.00E-01	1.01E-01	2.63E-01	1.45E+00
46	5.65E-02	3.22E-03	1.44E-02	3.25E-01	3.99E-01
47	4.87E-02	4.84E-03	9.58E-04	9.82E-03	6.43E-02
48	5.87E-01	8.24E-02	3.77E-02	5.28E-02	7.60E-01
49	5.87E-01	8.24E-02	3.77E-02	5.28E-02	7.60E-01
50	1.64E-01	3.65E-01	1.58E-01	1.54E-01	8.42E-01
51	3.07E-01	5.21E-01	1.40E-01	0.00E+00	9.68E-01
52	6.00E-01	4.13E-01	9.33E-02	1.67E-01	1.27E+00
53	4.38E-01	1.80E-01	3.97E-02	2.70E-01	9.29E-01
54	5.94E-01	2.85E-02	3.92E-03	2.36E-01	8.63E-01
55	4.38E-01	1.80E-01	3.98E-02	2.70E-01	9.29E-01
56	5.94E-01	2.85E-02	4.05E-03	2.36E-01	8.63E-01
57	7.62E-02	1.01E-01	6.77E-03	8.62E-03	1.93E-01

Continued on next page

sche_info

Continued from previous page

	#	PCA	Correlation	Chi-square	Info Gain	Sum
	58	6.50E-01	1.26E-03	8.78E-03	2.48E-01	9.08E-01
	59	5.55E-01	2.06E-01	3.31E-02	2.23E-01	1.02E+00
	60	1.56E-02	2.63E-02	1.50E-04	4.27E-02	8.48E-02
	61	1.17E-01	1.58E-01	1.36E-02	1.33E-02	3.02E-01
	62	1.99E-01	2.77E-01	4.01E-02	4.26E-02	5.59E-01
	63	6.00E-01	4.13E-01	9.33E-02	1.67E-01	1.27E+00
	64	6.00E-01	4.13E-01	9.33E-02	1.67E-01	1.27E+00
	65	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
	66	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
	67	4.76E-03	0.00E+00	3.00E-08	1.35E-05	4.78E-03
	68	2.03E-02	5.24E-02	6.80E-02	1.75E-01	3.16E-01
	69	2.19E-03	2.72E-02	1.00E-03	1.39E-01	1.70E-01
	70	2.21E-01	2.79E-01	4.11E-02	4.30E-02	5.85E-01
	71	4.89E-02	3.11E-02	6.29E-04	1.35E-01	2.15E-01
signal_info	72	2.69E-02	5.31E-02	1.29E-03	1.47E-01	2.29E-01
	73	9.32E-01	6.24E-01	7.76E-01	7.11E-01	3.04E+00
	74	7.32E-39	0.00E+00	0.00E+00	0.00E+00	7.32E-39
	75	1.75E-03	3.25E-03	6.23E-06	1.64E-02	2.14E-02
	76	2.13E-42	0.00E+00	0.00E+00	0.00E+00	2.13E-42
	77	4.06E-01	4.18E-01	2.83E-02	3.28E-01	1.18E+00
	78	1.26E-01	1.50E-01	8.51E-03	4.84E-01	7.69E-01
	79	5.48E-01	5.39E-01	5.56E-02	4.33E-01	1.58E+00
	80	5.31E-01	6.29E-01	2.08E-02	4.95E-01	1.68E+00
	81	2.24E-45	0.00E+00	0.00E+00	0.00E+00	2.24E-45
	82	3.07E-67	0.00E+00	0.00E+00	0.00E+00	3.07E-67
	83	6.12E-01	6.82E-01	1.37E-02	6.79E-01	1.99E+00
	84	4.85E-01	4.23E-01	1.93E-02	4.69E-01	1.40E+00
	85	3.94E-01	4.50E-01	7.55E-03	4.99E-01	1.35E+00
	86	2.43E-01	2.74E-01	7.55E-03	5.00E-01	1.02E+00
	87	2.70E-01	2.52E-01	1.50E-06	1.84E-01	7.06E-01
	88	3.21E-01	2.43E-01	1.93E-02	2.62E-01	8.46E-01
	89	5.92E-01	7.17E-01	3.18E-01	5.11E-01	2.14E+00
	90	5.74E-45	0.00E+00	0.00E+00	0.00E+00	5.74E-45
	91	1.78E-29	0.00E+00	0.00E+00	0.00E+00	1.78E-29
	92	1.12E-64	0.00E+00	0.00E+00	0.00E+00	1.12E-64
	93	4.28E-55	0.00E+00	0.00E+00	0.00E+00	4.28E-55

Continued on next page

Continued from previous page

	#	PCA	Correlation	Chi-square	Info Gain	Sum
	94	9.78E-05	2.48E-03	3.37E-06	8.34E-02	8.59E-02
	95	6.65E-01	8.18E-01	3.52E-01	5.12E-01	2.35E+00
	96	5.13E-01	5.69E-01	2.05E-01	5.21E-01	1.81E+00
	97	6.34E-70	0.00E+00	0.00E+00	0.00E+00	6.34E-70
	98	1.58E-02	1.22E-01	2.74E-02	1.77E-01	3.43E-01
	99	1.58E-02	1.22E-01	2.74E-02	1.77E-01	3.43E-01
	100	7.07E-01	1.82E-01	7.93E-01	7.10E-01	2.39E+00
	101	1.05E-01	5.80E-02	9.04E-03	1.97E-01	3.70E-01
	102	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
others	103	5.05E-03	4.38E-03	5.67E-06	8.03E-03	1.75E-02
	104	4.18E-02	2.92E-02	4.60E-04	9.01E-02	1.62E-01
	105	1.96E-71	0.00E+00	0.00E+00	0.00E+00	1.96E-71
	106	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
	107	5.58E-73	0.00E+00	0.00E+00	0.00E+00	5.58E-73
	108	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
	109	1.28E-02	8.56E-02	4.41E-03	5.33E-02	1.56E-01
	110	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
	111	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
	112	2.66E-18	0.00E+00	0.00E+00	0.00E+00	2.66E-18
	113	5.44E-17	0.00E+00	0.00E+00	0.00E+00	5.44E-17
	114	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00

Concluded

Chapter 5

Analysis of Experimental Results

This chapter introduces the experimental results of our design, including experimental configuration, experimental setup for data collection, experimental setup for data processing, experimental setup for classification, and results for local classifiers, parallel classifiers and enhanced neural network.

5.1 Experimental Configuration

5.1.1 Experimental Setup for Data Collection

To collect sufficient datasets, we deployed our data collection agents on 2 computers, each is installed with Linux Ubuntu 14.04 system is required to compile our custom Android system kernel modules. The module, constructed for Android 4.4 running on a Nexus 5 system [5], is compatible with different Android platforms. The Nexus Android system also offers authentic security and computational services for customers and developers. We set up two LG Nexus 5 smartphones to gather datasets and build our data storage pools to retrieve datasets automatically.

Data Storage With high availability and effective scalability, Apache Cassandra database [3] is installed to save massive data in parallel instead of a traditional relational schema. In our experiment, we configure Apache Cassandra version 2.1.11 for instance collection and data storage. Owing to the complexity of the data structure, the Cassandra of column indexes is a NoSQL alternative to compress redundant data records. We create a malware data table and a benign data table to synchronously store data records from the Android phones.

Data Collector In order to repeatedly collect Android data, we separate the operation into two steps: HTTP Insertion and UDP Listening. HTTP insertion creates a connection to

execute basic SQL statements and operate data transactions between Cassandra database and Android product. UDP Listening transits the data processing to background execution and tracks log files of IP addresses and port connections. The two steps simplify repetitively manual collectors by switching HTTP data operation and UDP operation listening when triggering the analysis signal. Here we set check constraints [20] to inspect the completeness and correctness of collected data samples in the Cassandra database.

5.1.2 Experimental Setup for Data Processing

Data PreProcessing We evaluate the four popular dimension reduction algorithms using by the open-source tool, RapidMiner [7], which is installed in an OS X Yosemite system of 3.5 GHz Intel Core i7, 16 GB DDR3, 1 TB hard disk, and 2GB Graphics. Considering the comparison of performance with our WBD technique, we also design a data preprocessing program in MATLAB R2015a for calculating variances after DCT transformation. The intermediate data results are saved temporally in the hard disk as the input data of the precise classification.

Data Instances The training dataset is comprised of 0.6M lines malware data records and 0.6M lines benign data records after removing the redundant records, where each line includes relevant features. Totally, we collect training data of 190 malware Android apps and 190 benign Android apps, taking 15K samples for each Android application. In contrast, the testing dataset contains 85 malware apps and 85 benign apps, each of them including 0.3M lines data records, respectively.

5.1.3 Experimental Setup for Classification

Firstly, we configure the machine learning environment for data training and testing in 2 machines, one has 3.5 GHz Intel Core i7, 16 GB DDR3, 1 TB hard disk, and 2GB Graphics and another with 2.6 GHz Intel Core i7, 8 GB DDR3, 758 GB hard disk and 4GB NVIDIA GeForce Graphics. Weka, RapidMiner and Clementine are different machine learning tools

Table 5.1: Hadoop Configurations

Parameter Name	Value
yarn.nodemanager.resource.memory-mb	22GB
yarn.nodemanager.vmem-pmem-ratio	6
yarn.scheduler.minimum-allocation-mb	1GB
yarn.app.mapreduce.am.resource.mb	2GB
yarn.scheduler.maximum-allocation-mb	2GB
mapreduce.map.memory.mb	2GB
mapreduce.reduce.memory.mb	2GB
mapreduce.map.java.opts	2GB
mapreduce.reduce.java.opts	2GB

Table 5.2: Spark Configurations

Parameter Name	Value
spark.master	spark://gpu-0-1:7077
spark.eventLog.enabled	true
spark.driver.memory	20 GB
spark.executor.uri	hdfs:///spark-1.6.0-hadoop2.6.tgz
MESOS_NATIVE_JAVA_LIBRARY	/local//lib/libmesos.so

in industrial data analysis. We set up the three tools in our local machine for identifying the malware instances from benign samples.

Our parallel experiments are executed on Apache Hadoop v2.6.0 and Apache Spark v1.6.0. The configurations of Hadoop are listed in Table 5.1, and the configurations of Spark are shown in Table 5.2. In addition, the Apache Mesos v0.27.1 [4] is used for managing Spark running and dispatching resources.

Furthermore, we evaluate the performance of EBP network on our IBM super computer **Cirrascale**. The super computer supports parallel computing with 48 CPU cores and 260 GB memory, which attributes to the in-memory calculation of the exascale data. Furthermore, in order to improve the performance, Nvidia Tesla K80 graphic cards are configured in our super computer. To simplify the evaluation process, we implement these algorithms with MATLAB R2016a and collect the Android application samples with Python programming language remotely.

5.2 Results of Local Classifiers

5.2.1 Distribution and Analysis of Kernel Features

Table 5.3 shows the distribution and analysis of the normalized weights of the 112 task kernel features with PCA, Correlation, Chi-square and Info Gain methods. PCA method can achieve 28 parameters (16 mem_info parameters, 8 signal_info parameters and 4 sche_info parameters) with high weights between 50% and 100%, 19 parameters (8 mem_info parameters, 8 signal_info parameters and 3 sche_info parameters) with weights between 10% and 50%, 36 parameters (16 mem_info parameters, 5 signal_info parameters, 6 sche_info parameters, 6 others and 2 task_state) with low weights between 0% and 10%, the rest 29 parameters with 0 % weights.

The correlation method analyzes the 112 features with a similar result, 20 parameters (13 mem_info parameters and 7 signal_info parameters) with high weights between 50% and 100%, 32 parameters (15 mem_info parameters, 10 signal_info parameters, 7 sche_info parameters) with weights between 10% and 50%, 26 parameters (12 mem_info parameters, 4 signal_info parameters, 5 sche_info parameters, 3 others and 2 task_state) with low weights between 0% and 10%, the rest 34 parameters with 0 % weights.

The Chi-square calculates the weights for 112 kernel features and achieves a distribution with minor difference, 11 parameters (9 mem_info parameters and 2 signal_info parameters) with high weights between 50% and 100%, 10 parameters (7 mem_info parameters and 3 signal_info parameters) with weights between 10% and 50%, 58 parameters (24 mem_info parameters, 16 signal_info parameters, 13 sche_info parameters, 3 others and 2 task_state) with low weights between 0% and 10%, the rest 33 parameters with 0 % weights.

The Info Gain method evaluates these 112 features and obtains the following results, 18 parameters (11 mem_info parameters and 7 signal_info parameters) with high weights between 50% and 100%, 49 parameters (26 mem_info parameters, 12 signal_info parameters, 7 sche_info, 2 others, and 2 task_state) with weights between 10% and 50%, 15 parameters

Table 5.3: Distribution of 112 Task Parameters Normalized Weights with PCA, Correlation, Chi-square and Info Gain Methods: **mem_info**, the most correlated feature set for classification, achieves the maximum number of large weights between 50% and 100% in 4 different techniques, next is **signal_info**, **sche_info**, **others** and **task_state** also contribute to precise classification. The details are located in Table 5.4.

		# of Param. 50% - 100%	# of Param. 10% - 50%	# of Param. 0 % - 10%	# of Param. 0 %	Total #
PCA	mem_info	16	8	16	8	48
	signal_info	8	8	5	9	30
	sche_info	4	3	6	2	15
	others	0	0	7	6	13
	task_state	0	0	2	4	6
	Total	**28**	**19**	**36**	**29**	**112**
Correlation	mem_info	13	15	12	8	48
	signal_info	7	10	4	9	30
	sche_info	0	7	5	3	15
	others	0	0	3	10	13
	task_state	0	0	2	4	6
	Total	**20**	**32**	**26**	**34**	**112**
Chi-square	mem_info	9	7	24	8	48
	signal_info	2	3	16	9	30
	sche_info	0	0	13	2	15
	others	0	0	3	10	13
	task_state	0	0	2	4	6
	Total	**11**	**10**	**58**	**33**	**112**
Info Gain	mem_info	11	26	2	9	48
	signal_info	7	12	2	9	30
	sche_info	0	7	6	2	15
	others	0	2	5	6	13
	task_state	0	2	0	4	6
	Total	**18**	**49**	**15**	**30**	**112**

(2 mem_info parameters, 2 signal_info parameters, 6 sche_info parameters, 5 others) with low weights between 0% and 10%, the rest 30 parameters with 0 % weights.

5.2.2 Comparison of newly infected and previously infected parameters

Figure 5.1: Comparison of Currently Infected Parameters and Previously Infected Parameters

Fig. 5.1 shows the analysis of newly infected parameters in Table 2.1 [1] and previously infected parameters in [75] of the Task Structure. All number denotes the parameter in Table 2.1 [1]. In the real experiment, we collected 112 parameters divided into 5 categories and added 26 new **mem_info** parameters, 26 new **signal_info** parameters, 11 new **sche_info** and 20 new **others** parameters, where 3 parameters have not used in the current Android and malware applications start to modify the newly-infected parameters. Note that we collected 112 task parameters, 80 out of them are newly collected, denoted as **newly-infected**, 32 out of them are previously used in [75], denoted as **previously-infected**.

Fig. 5.2 exhibits the weights' distribution of previously infected parameters. The x-axis value denotes the parameter's number and category in Table 5 [1] and y-axis denotes the value of normalized weights with PCA, Correlation, Chi-square and Info Gain methods. The previously-infected parameters are distributed in **mem_info** area, 6 out of 22

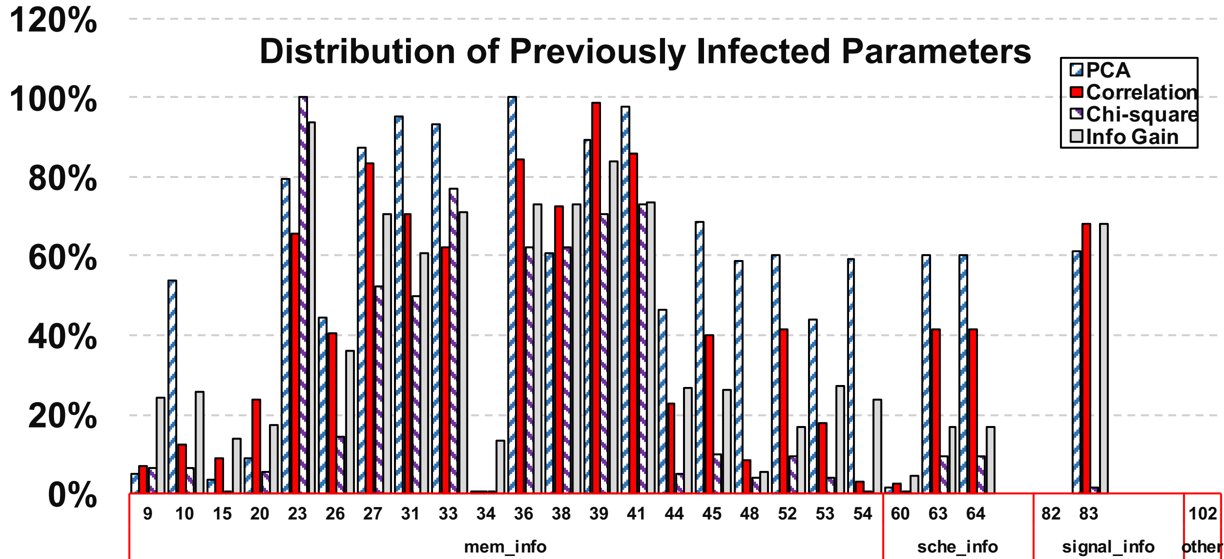

Figure 5.2: Non-Zero Normalized Weights of Previously-Infected Task Parameters (There are 32 previously-infected task parameters shown in Fig. 5.1 in detail.)

mem_info in Fig. 5.1 achieve large weights (above 50%). However, the old datasets lack enough **signal_info**, **sche_info**, **others** features.

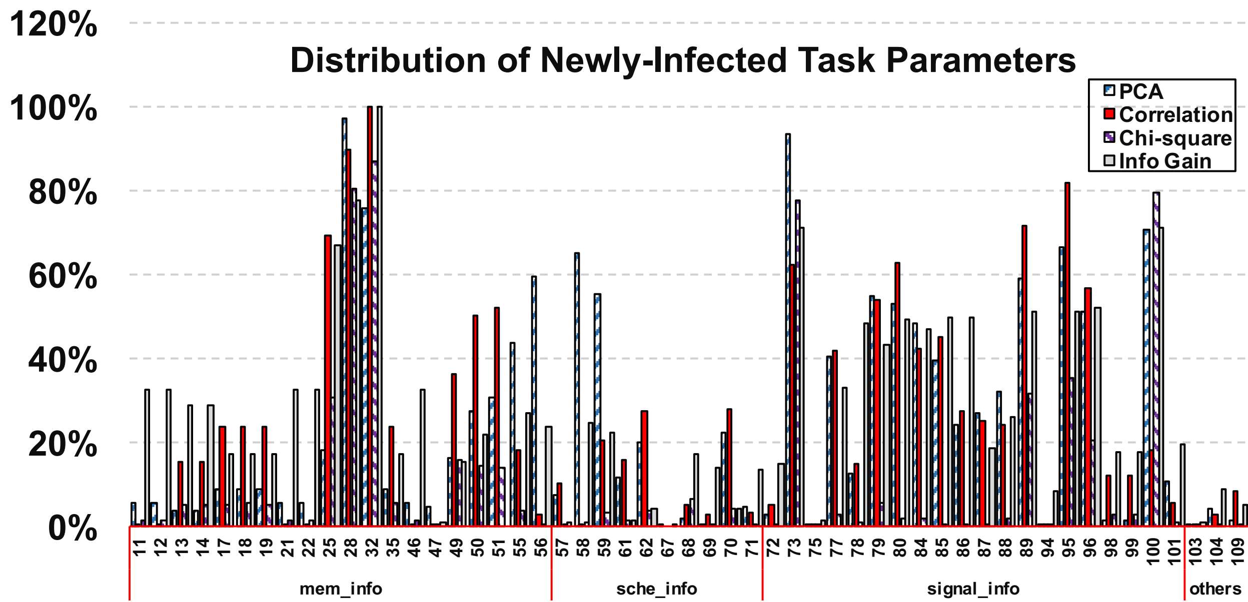

Figure 5.3: Non-Zero Normalized Weights of Newly-Infected Task Parameters (There are 80 newly-infected/currently infected task parameters shown in Fig. 5.1 in detail.)

Fig. 5.3 shows the weights' distribution of newly infected parameters. The x-axis value denotes the parameter's number and category in Table 5 [1] and y-axis denotes the value

of normalized weights with PCA, Correlation, Chi-square and Info Gain methods. 20 new **mem_info** parameters, 10 new **sche_info**, 20 new **signal_info** parameters and 3 new **others** achieve different weights impacting on selection of correlated features. The weights of rest parameters equals to 0. Among newly-infected parameters, **signal_info** retains more parameters with large weights than others, and **mem_info** also attains several parameters with large weights.

5.2.3 Cross-Validation Results

Comparison between WBD and VBD

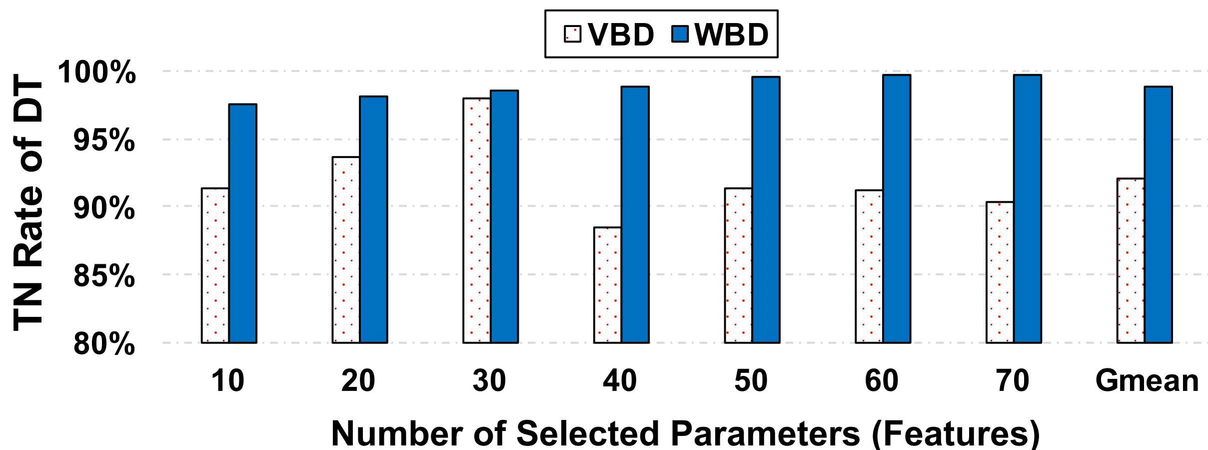

Figure 5.4: True Negative Rate by Decision Tree With the Increasing Number of Selected Features: VBD is proposed in [75] and WBD denotes our methods, on average WBD achieves 6% improvement of TN.

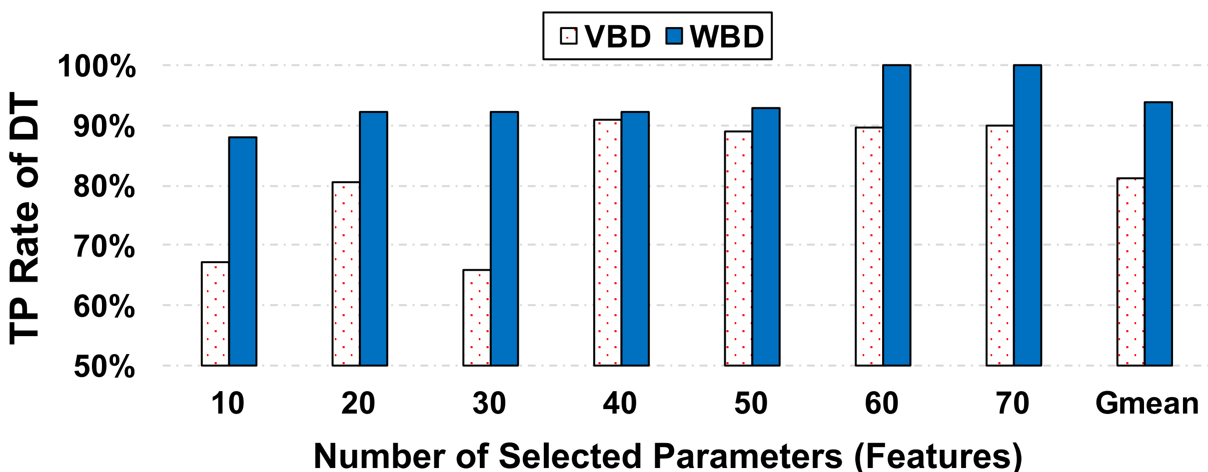

Figure 5.5: True Positive Rate by Decision Tree With the Increasing Number of Selected Features: VBD is proposed in [75] and WBD denotes our methods, on average WBD achieves 12% improvement of TP.

Fig. 5.4 shows the True Negative (TN) rate with Decision Tree technique for VBD and WBD. Since the VBD authors only provide Decision Tree Classifier in their paper, we compare our Decision Tree results with VBD. The X-axis denotes the number of selected

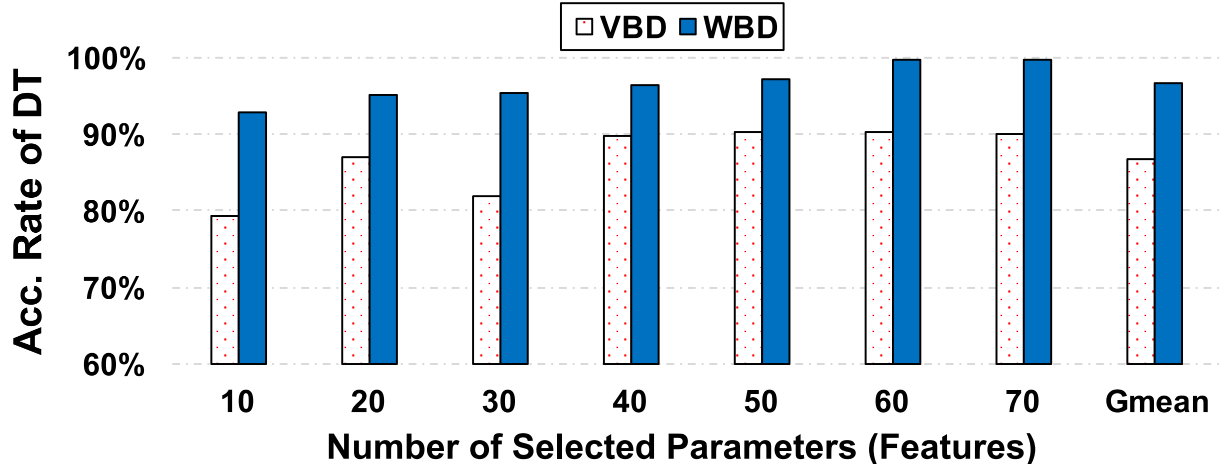

Figure 5.6: Accuracy Rate by Decision Tree With the Increasing Number of Selected Features: VBD is proposed in [75] and WBD denotes our methods, on average WBD achieves 10% improvement of accuracy.

features among 112 attributes and the Y-axis is the TN rate. We can see WBD and VBD averagely achieve 98%, 92% TN rate, respectively. With the increasing of selected features' number, TN rate increases gradually both in WBD and VBD, but VBD's TN rate is lower than WBD's.

As the TP rate is the measurement of positive proportion, we evaluate TP rate separately in Fig. 5.5. On average WBD conserves 94% of TP rate compared to VBD (82%). TP rates from 10 to 70 selected features reveal the ascending trends as the same as the TN rate for WBD and VBD. When we train the data samples by taking 10 features, VBD leads to a lowest TP rate (68%). Then its TP rate has a dramatic increase (80%) by training 20 features and float slightly in the following tests. In contrast, WBD causes small-scale variation (88%-98%) as the changes of the features' number.

Fig. 5.6 further shows the accuracy rate between WBD and VBD. WBD preserves 97% of accuracy rate on average and VBD achieves 87% of accuracy rate. That is because cumulative variance of VBD destroys the regular pattern of interior data. In general, dimensional reduction incurs slow degradation of the accuracy rate in Fig. 5.6. Nevertheless, data manipulation inside each dimension leads to 3% -4% reduction of the accuracy rate in cross-validation tests.

Naive Bayes Results

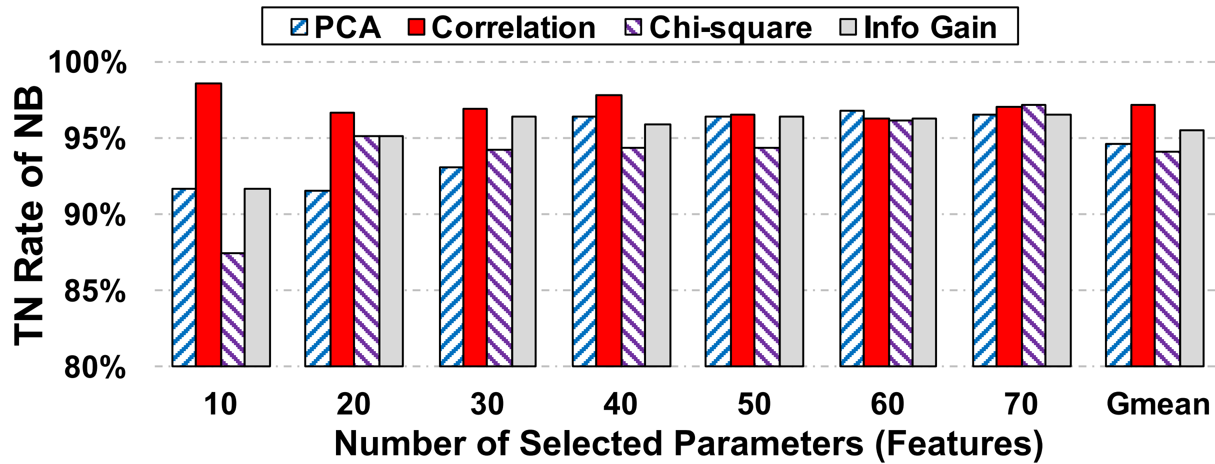

Figure 5.7: True Negative Rate by Naive Bayes Kernel With the Increasing Number of Selected Features: Correlation method leads to the highest TN than PCA, Chi-square, and Info Gain on average.

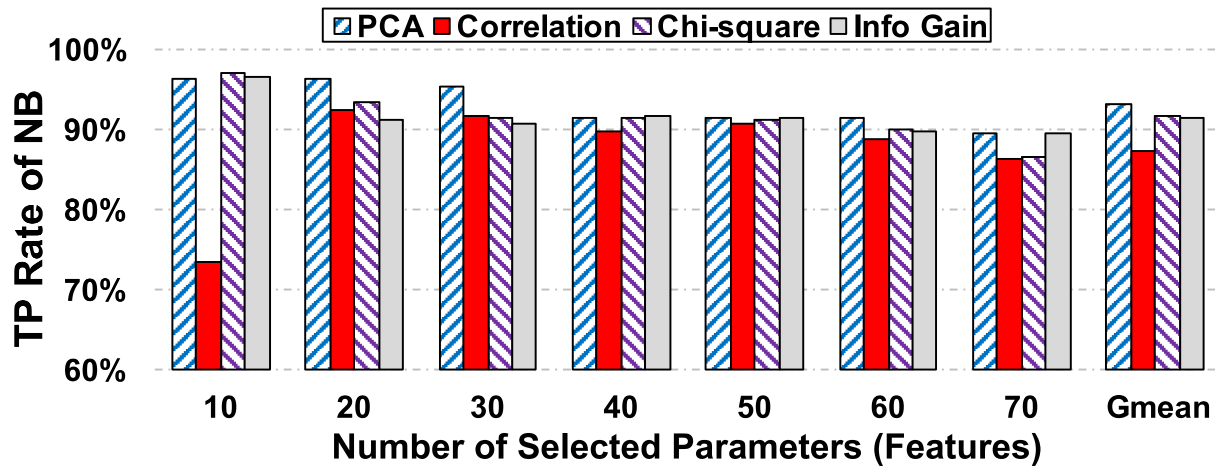

Figure 5.8: True Positive Rate by Naive Bayes Kernel With the Increasing Number of Selected Features: PCA achieves the best TP compared to others on average.

Fig. 5.7 shows TN rates of Naive Bayes Kernel Classifier along with the variation of the number of selected features. Naive Bayes Kernel Classifier achieves gradual increase of TN rate while selecting more features with PCA, Chi-square, and Info Gain. However, the classifier using Corrlelation leads to 97% of TN rate on average which is the highest among the four techniques of dimension reduction. Unlike Chi-square resulting in lower TN rate,

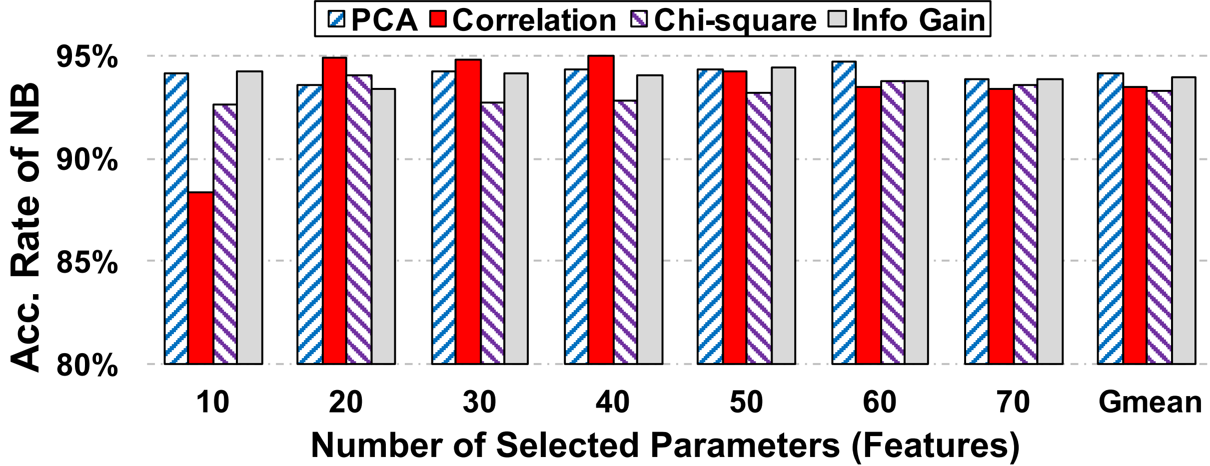

Figure 5.9: Accuracy Rate by Naive Bayes Kernel With the Increasing Number of Selected Features: 4 methods achieves the similar accuracy results on average, PCA achieves slightly higher accuracy.

Correlation and Info Gain save as much as 96% of TN rate. Furthermore, PCA also achieves 94% of TN rate compared to Chi-square.

Fig. 5.8 shows TP rates of Naive Bayes Kernel using dimension reduction techniques of PCA, Correlaltion, Chi-square, and Info Gain. PCA, Chi-square and Info Gain achieve 93%, 91%, and 91% of TP rates on average, but Correlation causes the lowest TP rate (87%) due to discrepancy in feature selection. Interestingly, TP rates of PCA, Chi-square and Info Gain decrease slightly along with the increase of the number of selected features, which means the memory attributes benefit malware application identification since the majority of preferential features is from **mem_info** descriptor of Table 2.1.

The enhancement of classification precision is ascribed to the feature selection in high-dimension dataset. Fig. 5.9 shows the accuracy rates of PCA, Correlation, Chi-square and Info Gain. PCA preserves 94.2% of accuracy rate compared to Correlation (93.4%) irregularly varying with the number of features. In contrast, Chi-square and Info Gain achieve 93.3%, 93.9% accuracy rate, respectively.

Decision Tree Results

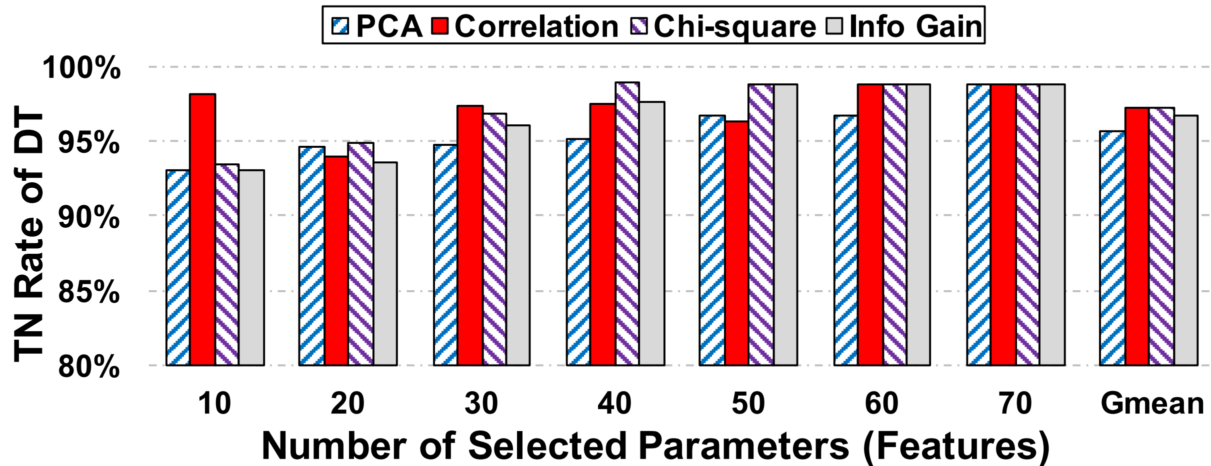

Figure 5.10: True Negative Rate by Decision Tree With the Increasing Number of Selected Features: Correlation and Chi-square methods lead to the highest TN than PCA and Info Gain.

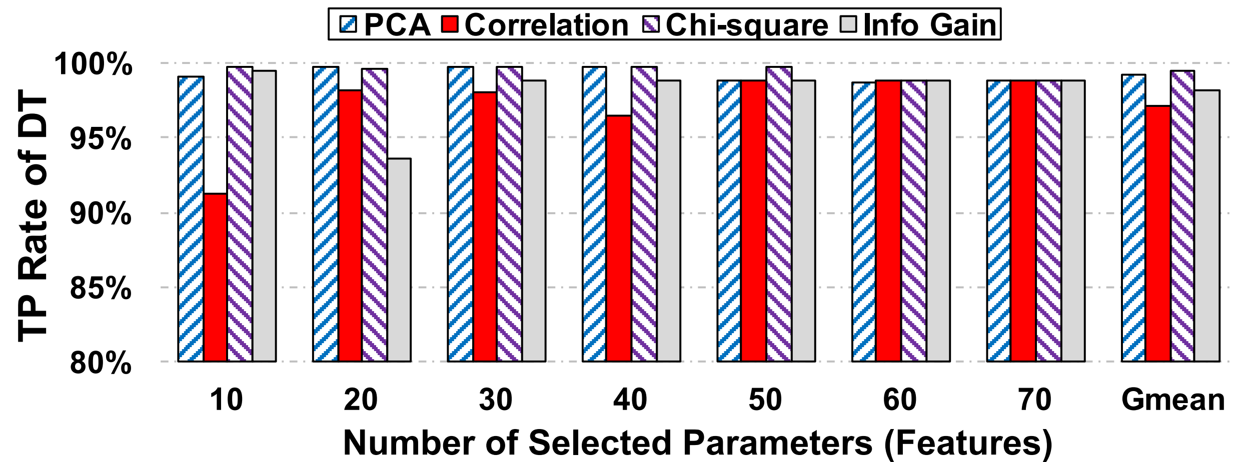

Figure 5.11: True Positive Rate by Decision Tree With the Increasing Number of Selected Features: Chi-square method achieves the best TP compared to others on average.

Fig. 5.10 shows results of TN rate of Decision Tree with PCA, Correlation, Chi-square and Info Gain. As we can see from Fig. 5.10, Correlation and Chi-square lead to 97% of TN rate in contrast with PCA (95.6%) and Info Gain (96.6%). However, Chi-square demonstrates the slow growth of TN rates with the increase of feature number, while Correlation maintains the relatively stable status. Similarly, PCA and Info Gain also cause the TN rate's

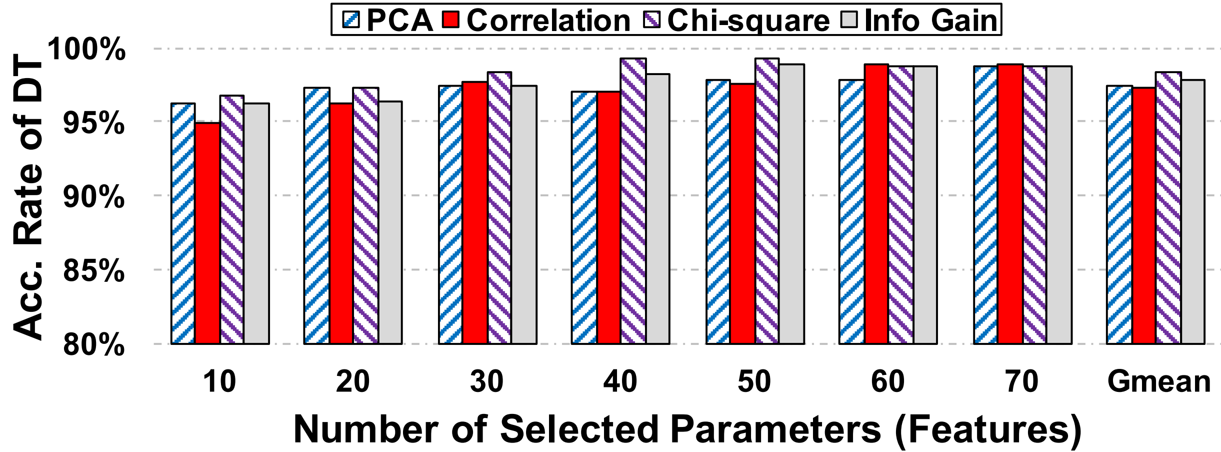

Figure 5.12: Accuracy Rate by Decision Tree With the Increasing Number of Selected Features: 4 methods achieve the similar accuracy results on average, Chi-square can achieve a bit higher accuracy.

increment from 93% (10 features) to 98% (70 features). Overall Decision Tree is a better classifier than Naive Bayes and achieves higher average TN rate.

From Fig. 5.11, we can see PCA and Chi-square achieve as much as 99% of TP rate on average. Meanwhile, Correlation and Info Gain also on average achieve 97%, 98% of TP rates, respectively. For Decision Tree, the features selected by the four techniques conduce to distinguish benign behaviors from the union of malware and non-malware samples. TP rates of PCA and Chi-square increase lightly between 98% and 99%. However, TP rates of Correlation and Info Gain demonstrates unexpected increment or decline.

Fig. 5.12 illustrates the overall accuracy rate of Decision Tree with PCA, Correlation, Chi-square and Info Gain. To be specific, PCA, Correlation, Chi-square and Info Gain achieve 97.4%, 97.3%, 98.4% and 97.8% of accuracy rate, respectively. For Decision Tree classifier, Chi-square leads to the most accurate classification results than PCA, Correlation and Info Gain.

Neural Network Results

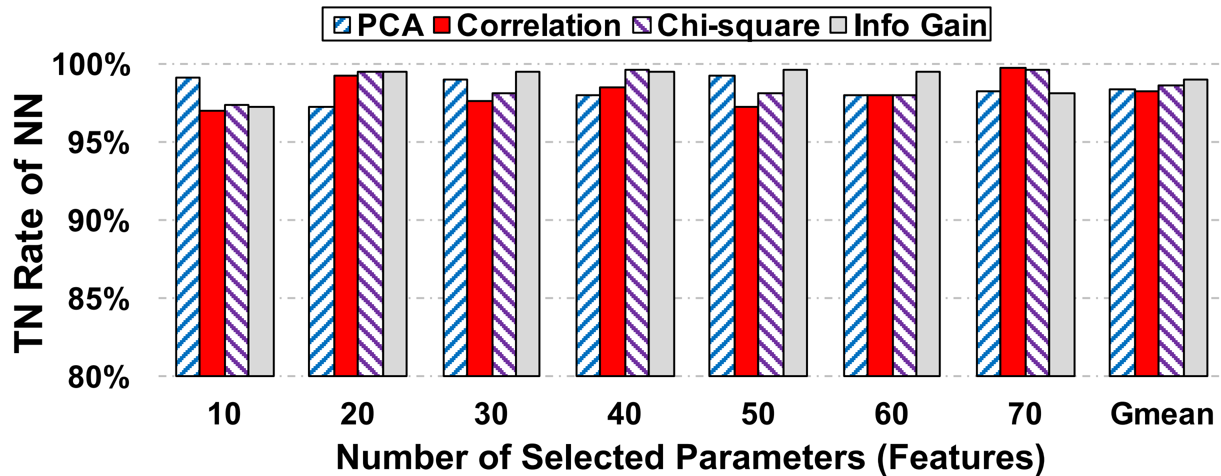

Figure 5.13: True Negative Rate by Neural Net With the Increasing Number of Selected Features: Info Gain method leads to the highest TN than PCA, Correlation, and Chi-square.

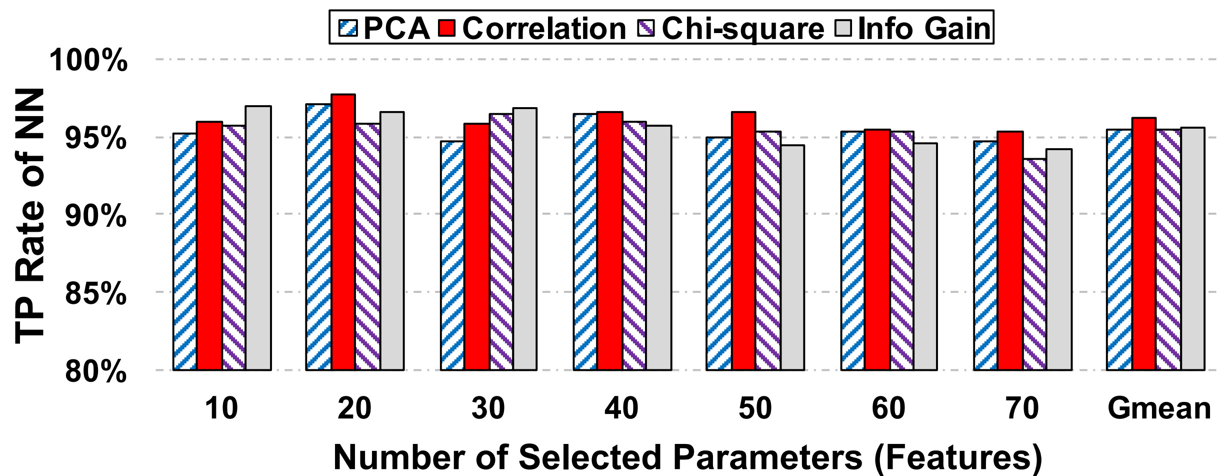

Figure 5.14: True Positive Rate by Neural Net With the Increasing Number of Selected Features: Correlation method achieves the best TP compared to others on average.

Fig. 5.13 shows the results of TN rate of Neural Network with PCA , Correlation, Chi-square and Info Gain. We can see PCA, Correlation, Chi-square and Info Gain cause very high TN rate (above 98%) on average, compared to Naive Bayes and Decision Tree. Due to nonlinear mapping of training models in Neural Network, in contrast with Decision Tree and Naive Bayes, PCA, Chi-square and Info Gain incur more accurate results, regardless the number of features.

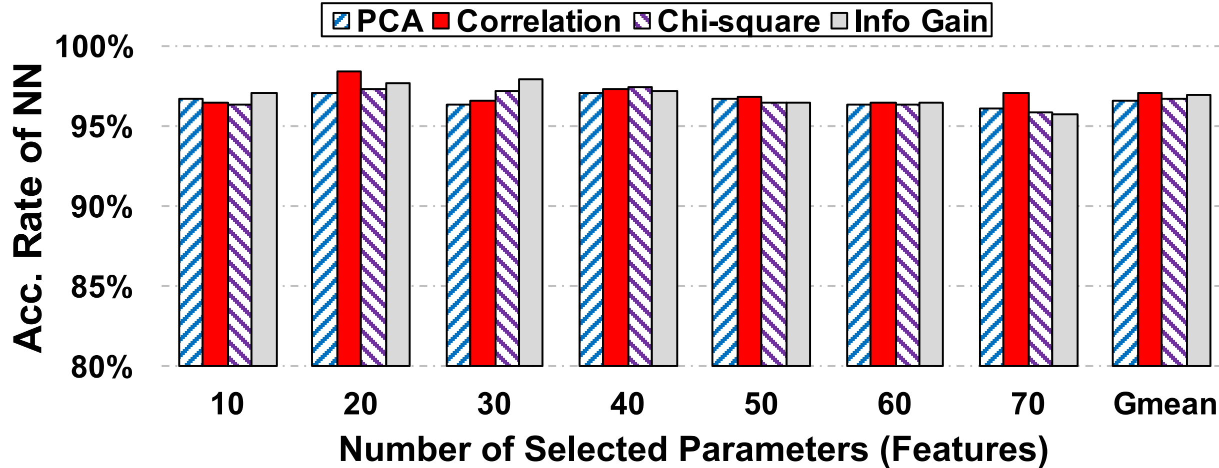

Figure 5.15: Accuracy Rate by Neural Net With the Increasing Number of Selected Features: 4 methods achieves the similar accuracy results on average, Correlation can achieve slightly higher accuracy.

Fig. 5.14 illustrates the TP rate of Neural Network with PCA, Correlation, Chi-square and Info Gain. As we can see, for PCA, Chi-square and Info Gain, Neural Network classifier achieves 95% of TP rate on average. Correlation has the best TP rate while selecting 20 features and leads to the highest overall TP rate compared to PCA, Chi-square and Info Gain. Although Neural Network classifier preserves better FN rate than Decision Tree, TP rate is 2-3% lower than Decision Tree in consideration of interaction of hidden layers.

Fig. 5.15 shows the accuracy of Neural Network with PCA, Correlation, Chi-square and Info Gain. As we can see, PCA, Correlation, Chi-square and Info Gain leads to 96.6%, 97.0%, 96.7% and 96.9% on average of the overall accuracy rate, respectively. Specifically, for Neural Network, when selecting as much as 60 features the accuracy rate approximately approaches the best prediction of malware and non-malware apps. However, 30 or 40 features are sufficient to calculate the weights of each neurons for precisely classifying two categories of data samples.

Experimental Results of NB, DT, NN

From Table 5.4, we can see Naive Bayes classifier preserves the lower precision compared to Decision Tree and Neural Network. Although Decision Tree leads to a much better

Table 5.4: TP Rate, TN Rate and Accuracy Rate According to Select Different Numbers of Features by PCA, Correlation, Chi-square and Info Gain with 3 Different Machine Learning Algorithms (Decision Tree, Naive Bayes and Neural Network)

Classifiers			#. of Selected Param.							
			10	20	30	40	50	60	70	Gmean
Decision Tree	PCA	TP Rate	99.14%	99.7%	99.71%	99.71%	98.82%	98.72%	98.83%	99.23%
		TN Rate	93.07%	94.64%	94.77%	95.14%	96.76%	96.76%	98.82%	95.69%
		Acc. Rate	96.30%	97.31%	97.375%	97.06%	97.765%	97.77%	98.82%	97.48%
	Correlation	TP Rate	91.25%	98.18%	98.09%	96.49%	98.82%	98.83%	98.83%	97.18%
		TN Rate	98.11%	94.03%	97.34%	97.49%	96.31%	98.83%	98.83%	97.26%
		Acc. Rate	94.88%	96.20%	97.72%	96.99%	97.60%	98.83%	98.83%	97.28%
	Chi-square	TP Rate	99.71%	99.60%	99.70%	99.71%	99.71%	98.83%	98.83%	99.44%
		TN Rate	93.45%	94.85%	96.91%	98.94%	98.79%	98.82%	98.82%	97.20%
		Acc. Rate	96.79%	97.35%	98.36%	99.33%	99.26%	98.82%	98.82%	98.38%
	Info Gain	TP Rate	99.47%	93.63%	98.83%	98.83%	98.83%	98.83%	98.83%	98.16%
		TN Rate	93.04%	93.63%	96.07%	97.64%	98.82%	98.82%	98.82%	96.66%
		Acc. Rate	96.26%	96.37%	97.50%	98.25%	98.83%	98.82%	98.82%	97.83%
Naive Bayes	PCA	TP Rate	96.43%	96.43%	95.31%	91.39%	91.38%	91.41%	89.40%	93.07%
		TN Rate	91.63%	91.54%	93.02%	96.32%	96.34%	96.74%	96.55%	94.56%
		Acc. Rate	94.15%	93.58%	94.20%	94.34%	94.35%	94.67%	93.89%	94.16%
	Correlation	TP Rate	73.39%	92.53%	91.58%	89.78%	90.77%	88.78%	86.21%	87.35%
		TN Rate	98.61%	96.59%	96.86%	97.79%	96.49%	96.24%	97.06%	97.09%
		Acc. Rate	88.39%	94.92%	94.83%	95.00%	94.26%	93.43%	93.34%	93.43%
	Chi-square	TP Rate	97.04%	93.41%	91.38%	91.47%	91.27%	90.04%	86.69%	91.57%
		TN Rate	87.36%	95.06%	94.25%	94.31%	94.31%	96.15%	97.16%	94.04%
		Acc. Rate	92.62%	94.08%	92.75%	92.82%	93.16%	93.76%	93.58%	93.25%
	Info Gain	TP Rate	96.63%	91.18%	90.77%	91.60%	91.47%	89.78%	89.42%	91.52%
		TN Rate	91.66%	95.09%	96.37%	95.84%	96.34%	96.27%	96.56%	95.43%
		Acc. Rate	94.27%	93.33%	94.17%	94.04%	94.38%	93.77%	93.90%	93.98%
Neural Network	PCA	TP Rate	95.19%	97.04%	94.65%	96.49%	95.01%	95.31%	94.68%	95.48%
		TN Rate	99.09%	97.24%	99.00%	98.00%	99.25%	98.02%	98.20%	98.40%
		Acc. Rate	96.71%	97.12%	96.34%	97.10%	96.66%	96.39%	96.06%	96.62%
	Correlation	TP Rate	95.98%	97.76%	95.87%	96.54%	96.60%	95.49%	95.35%	96.22%
		TN Rate	97.03%	99.22%	97.62%	98.46%	97.26%	98.01%	99.76%	98.19%
		Acc. Rate	96.41%	98.35%	96.57%	97.31%	96.87%	96.49%	97.06%	97.01%
	Chi-square	TP Rate	95.69%	95.89%	96.50%	95.94%	95.28%	95.31%	93.60%	95.45%
		TN Rate	97.42%	99.52%	98.15%	99.57%	98.10%	98.02%	99.65%	98.63%
		Acc. Rate	96.38%	97.32%	97.17%	97.37%	96.40%	96.39%	95.89%	96.70%
	Info Gain	TP Rate	96.96%	96.56%	96.89%	95.73%	94.42%	94.61%	94.25%	95.63%
		TN Rate	97.19%	99.45%	99.43%	99.49%	99.64%	99.50%	98.05%	98.96%
		Acc. Rate	97.06%	97.71%	97.91%	97.21%	96.42%	96.50%	95.73%	96.93%

accuracy rate than Naive Bayes and Neural Network, Neural Network can achieve 2-3% higher TN rate than Decision Tree due to its accurate computational model. That means Neural Network classifier is more sensitive to recognizing the malicious apps as malware. Furthermore the methods of dimensional reduction cause minor changes (2%-3%) of accuracy rates when we select at least half of all features, note that in our experiment the total number of features is 112 and the half of them is around 60 features.

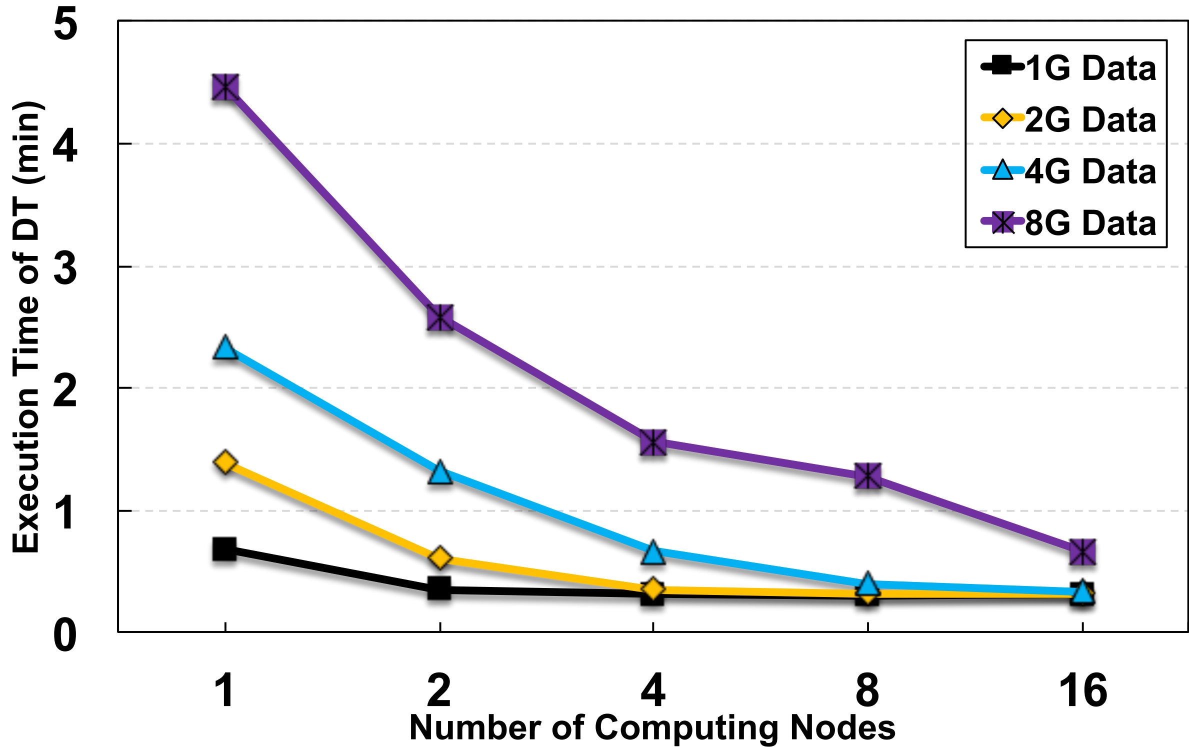

Figure 5.16: Execution Time (min) of DT Classifier

5.3 Results of Parallel Classifiers

5.3.1 Execution Time

Fig. 5.16, 5.17, 5.18 and 5.19 show execution times of DT, LR, SVM and NB classifiers with different computing nodes. The execution time of the four classifiers with a small dataset (1GB Dataset) does not change much while increasing the number of computing nodes (workers), since only a computing node with 12 cores and 22GB memory can process 1GB data computation. Obviously, 1 additional computing node increases gradually the overhead time with the increase of the size of datasets from 1GB to 8GB. However, for larger size of dataset (e.g., 4GB and 8GB), more computing nodes can quickly reduce the processing time.

From our experimental results and spark configuration, we can see 1 computing node (worker) can complete the computation of 1GB dataset. Therefore, 2GB dataset can be processed by 2 computing nodes, denoted as: 2GB $\Rightarrow$ 2 workers, similarly, 4GB $\Rightarrow$ 4 workers,

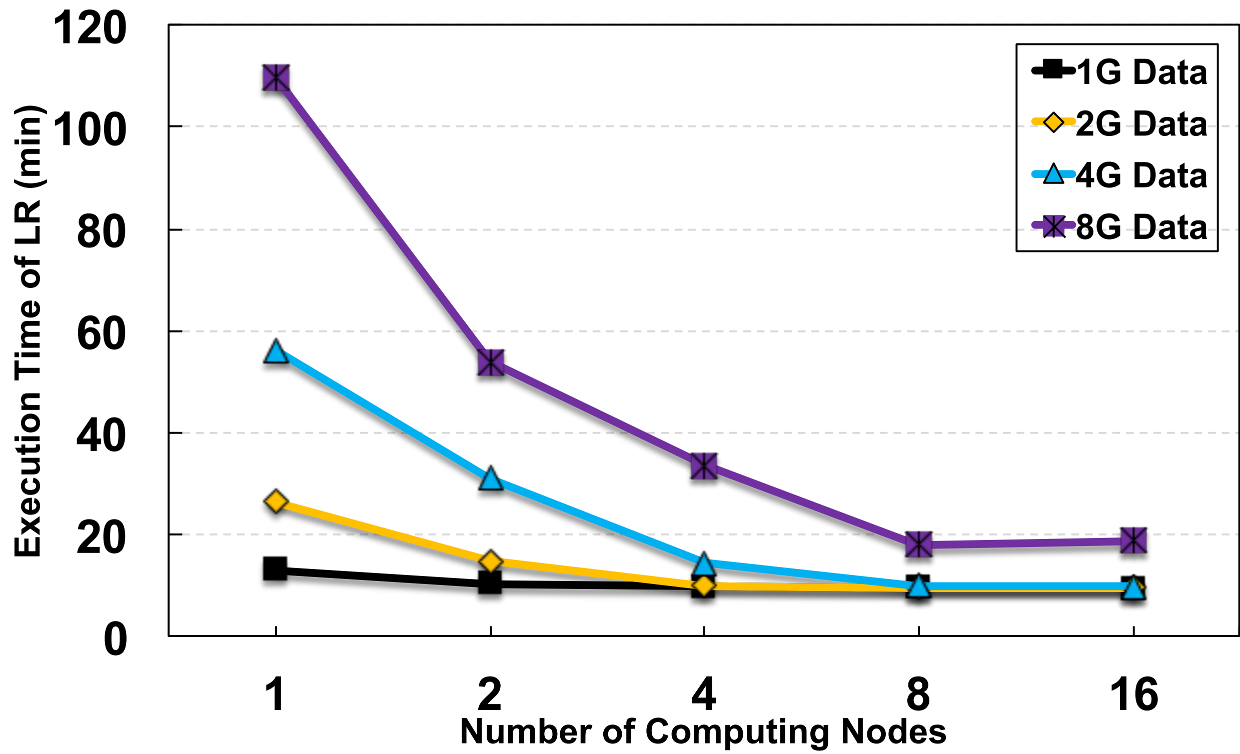

Figure 5.17: Execution Time (min) of LR Classifier

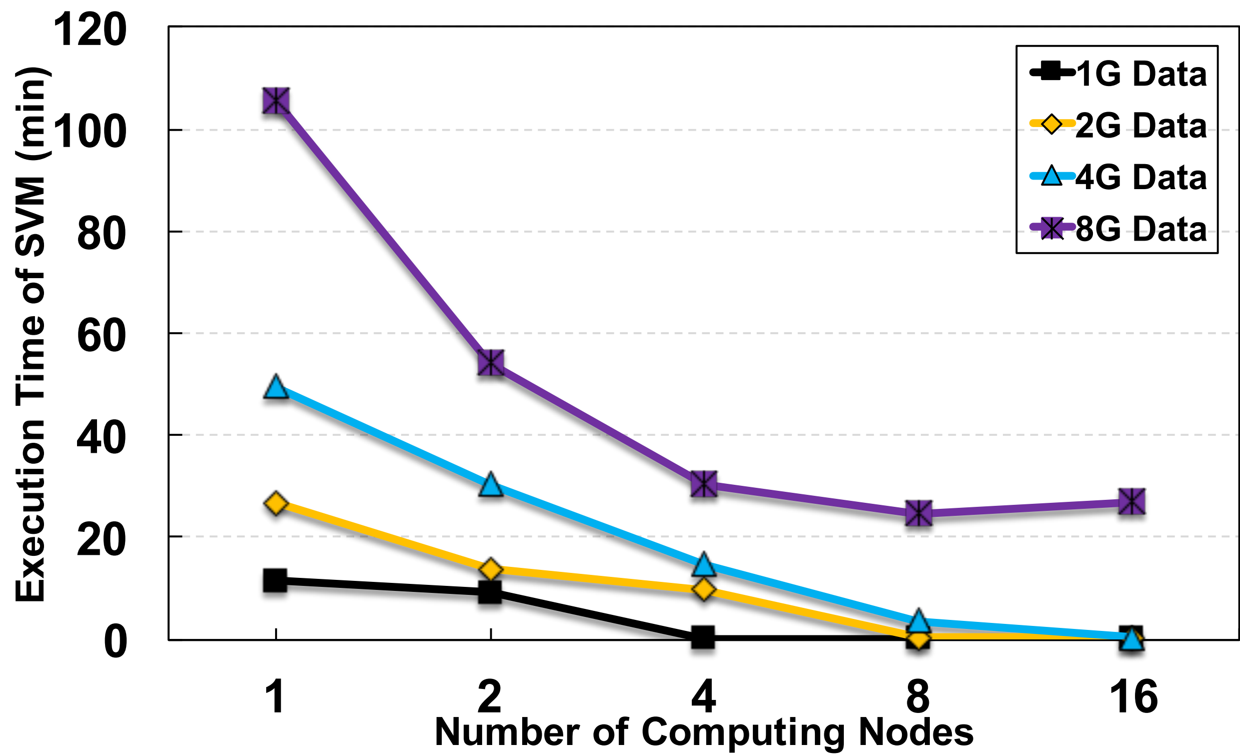

Figure 5.18: Execution Time (min) of SVM Classifier

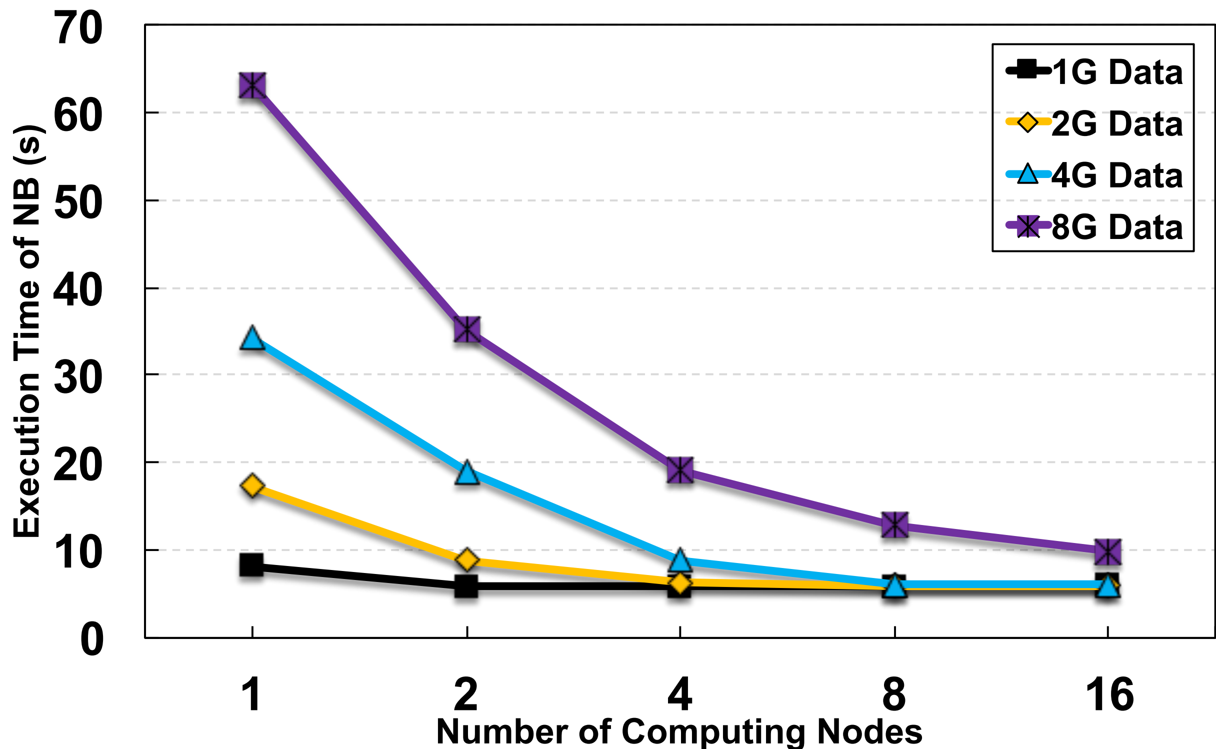

Figure 5.19: Execution Time (s) of NB classifier

and 8GB ⇒ 8 workers. Taking 8GB dataset as an example, the DT classifier decreases the execution time by 0.5X from 1 worker to 16 workers, LR, SVM and NB classifiers do likewise. However, the increase of number of workers also introduces extra overhead, such as scheduling policies, memory management, etc., causing slight increment of execution time when reaching for stable status.

The four classifiers show different performance results in terms of execution time. The SVM and LR classifiers in Fig. 5.17 and 5.18 are of the same order of magnitude for execution time due to iterations of linear computation. The DT classifier in Fig. 5.16 preserves a higher performance with less execution time. In contrast, the NB classifier in Fig. 5.19 performs the prediction with the least execution time.

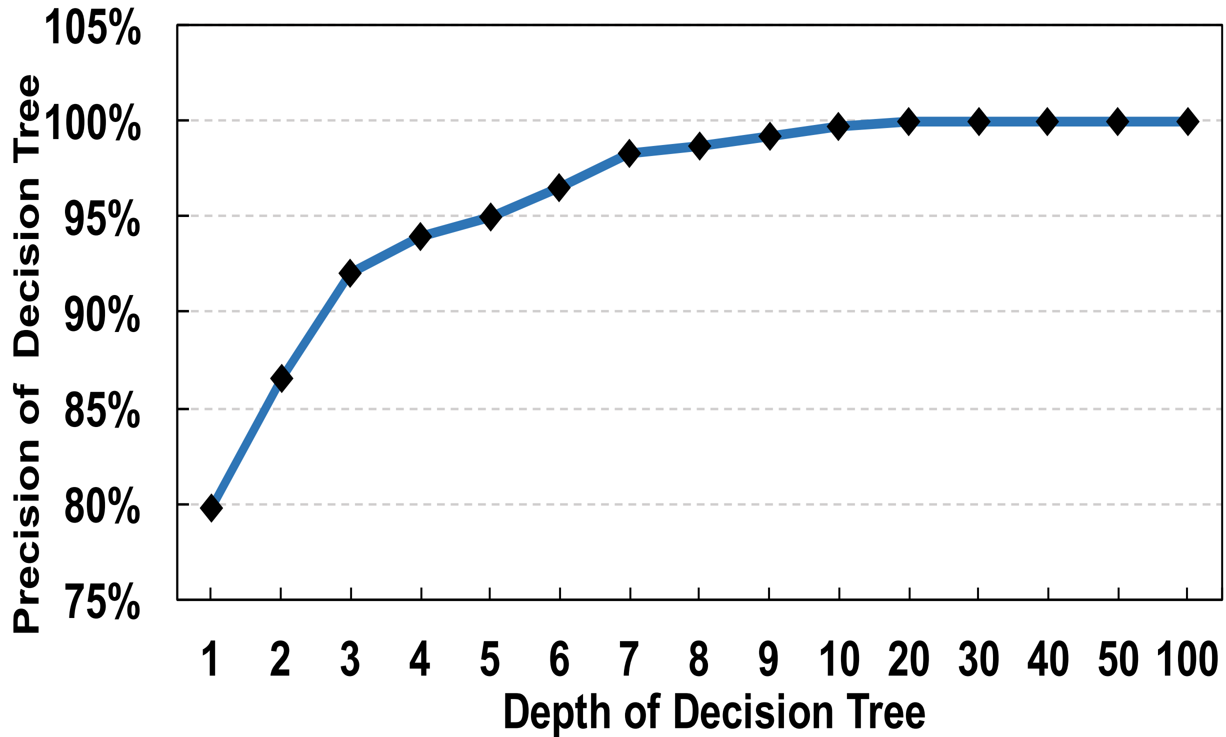

Figure 5.20: Classification Precision by DT Classifier

5.3.2 Classification Precision

Fig. 5.20 shows the classification precision by a DT classifier with the increase of the depth of the decision tree. As shown in Fig. 5.20, the precision can reach approximately 100% when increasing the depth of the decision tree to 20. Considering effectiveness of decision tree classification, the small depth number, e.g., 5, also can achieve a good accuracy rate with less computation overhead.

Fig. 5.21 shows the classification precision by a LR classifier with the increase of the number of iterations. We can see precision approaches to 95% after 3 iterations. Although the number of iterations rises to 100, the classification precision changes slightly between 0% and 1%.

Fig. 5.22 shows the classification precision by a SVM classifier with the increase of the number of iterations. We can see this result is similar to the LR and the precision approaches

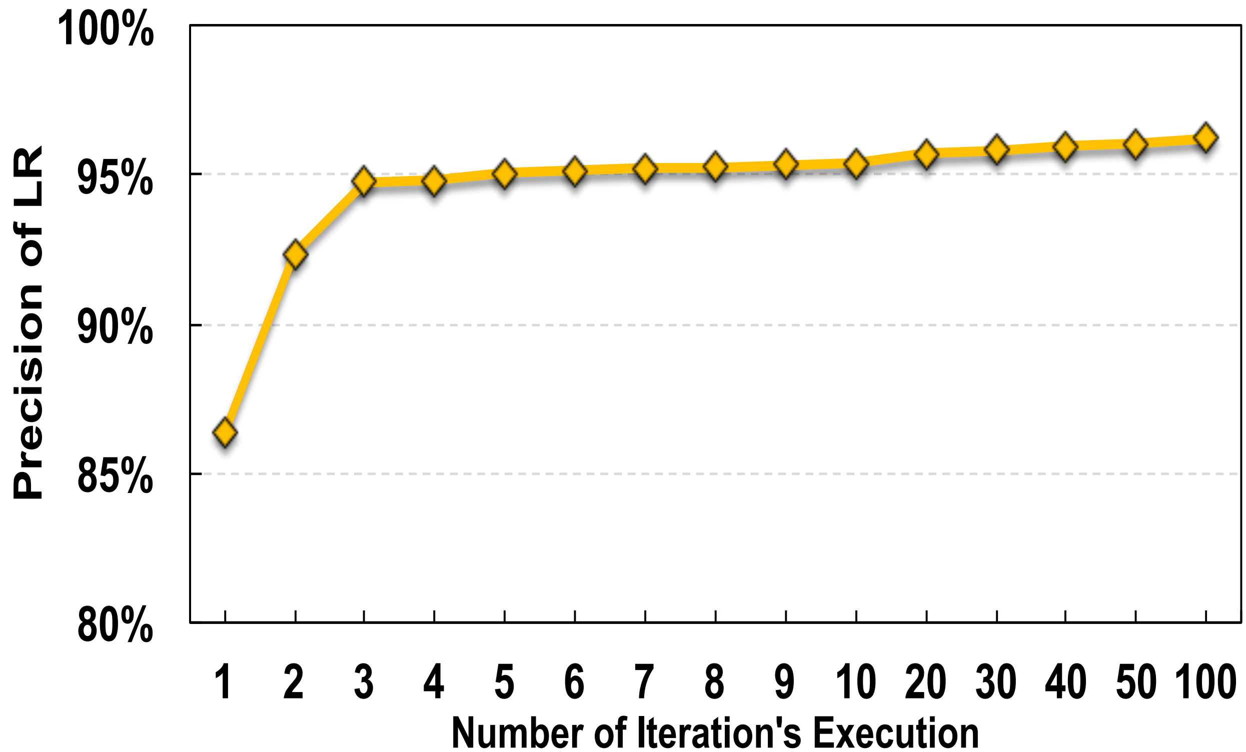

Figure 5.21: Classification Precision by LR Classifier

to 95% after 3 iterations. While the number of iterations rises to 100, the classification precision changes lightly.

Fig. 5.23 shows the classification precision by a NB classifier with selecting different number of samples. The precision does not vary along with the number of samples from 1 to 100. Obviously, NB classifier achieves a lower precision (82.5%) compared to DT classifier (above 95%), LR and SVM classifiers (95%-96%).

Fig. 5.24 shows the precision comparisons among the four classifiers. The DT, NB, LR and SVM classifiers preserve 99%, 83%, 96% and 96% precision, respectively. The DT classifier can achieve a better precision with less computation overhead compared to the other three classifiers.

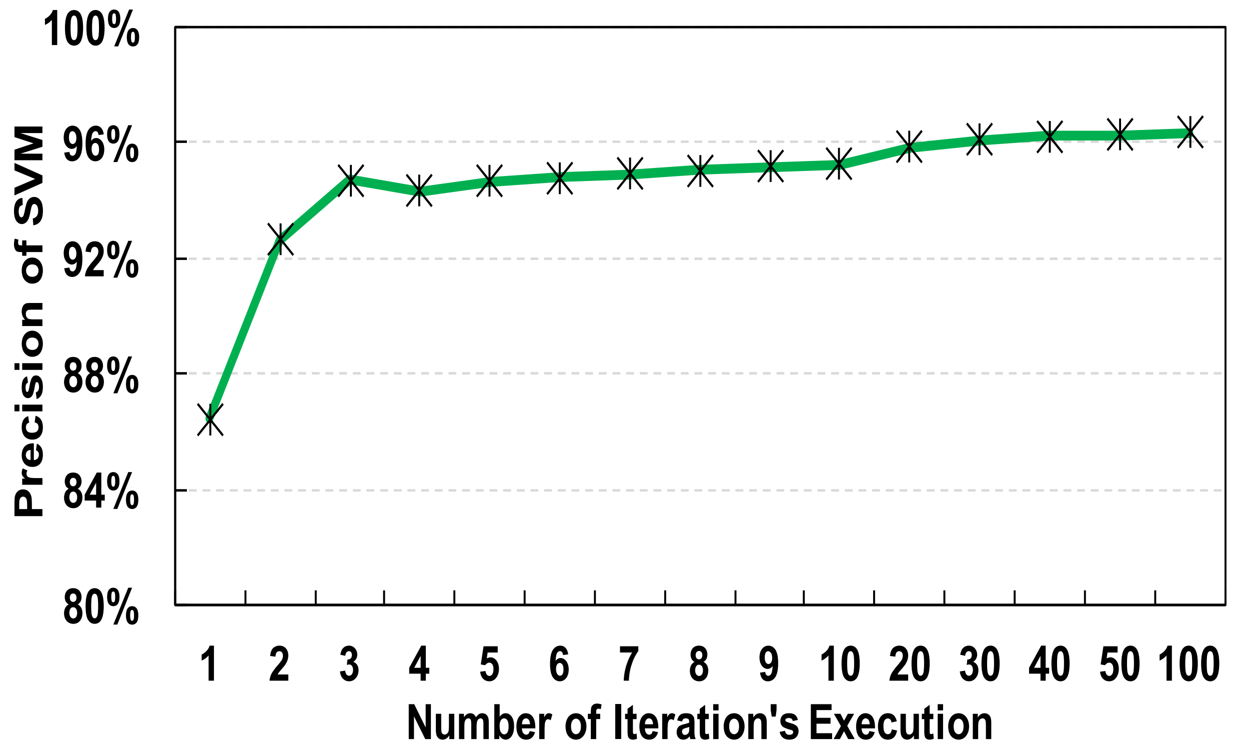

Figure 5.22: Classification Precision by SVM Classifier

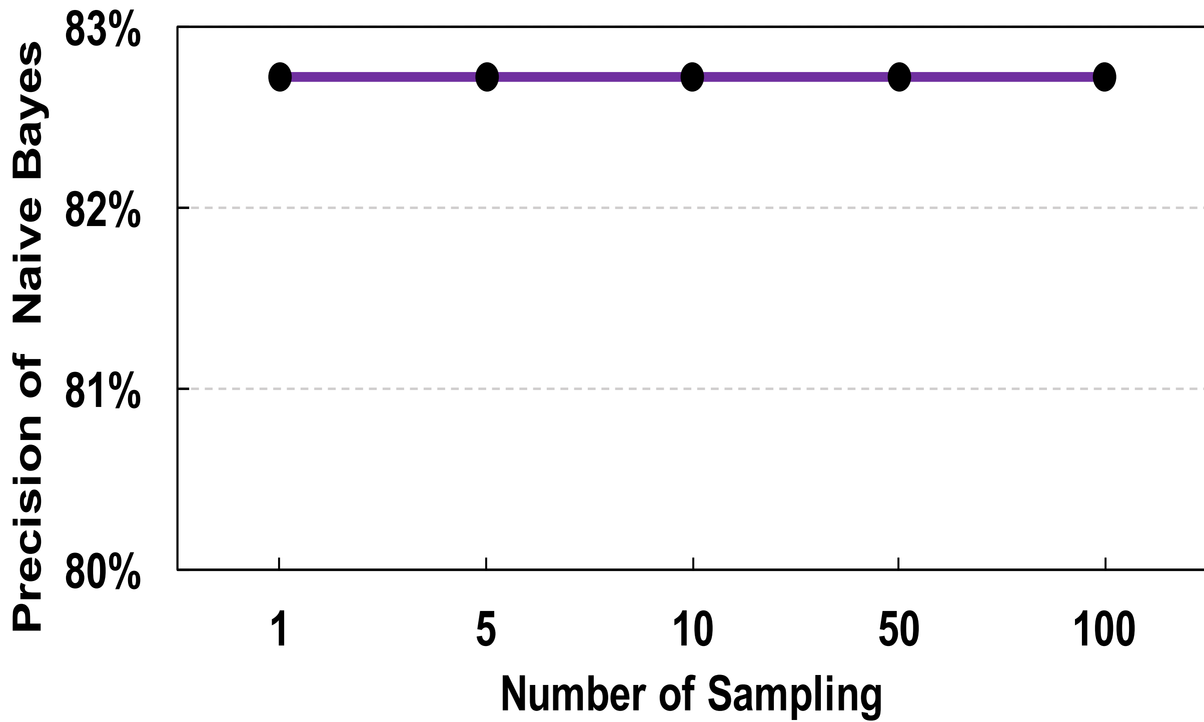

Figure 5.23: Classification Precision by NB Classifier

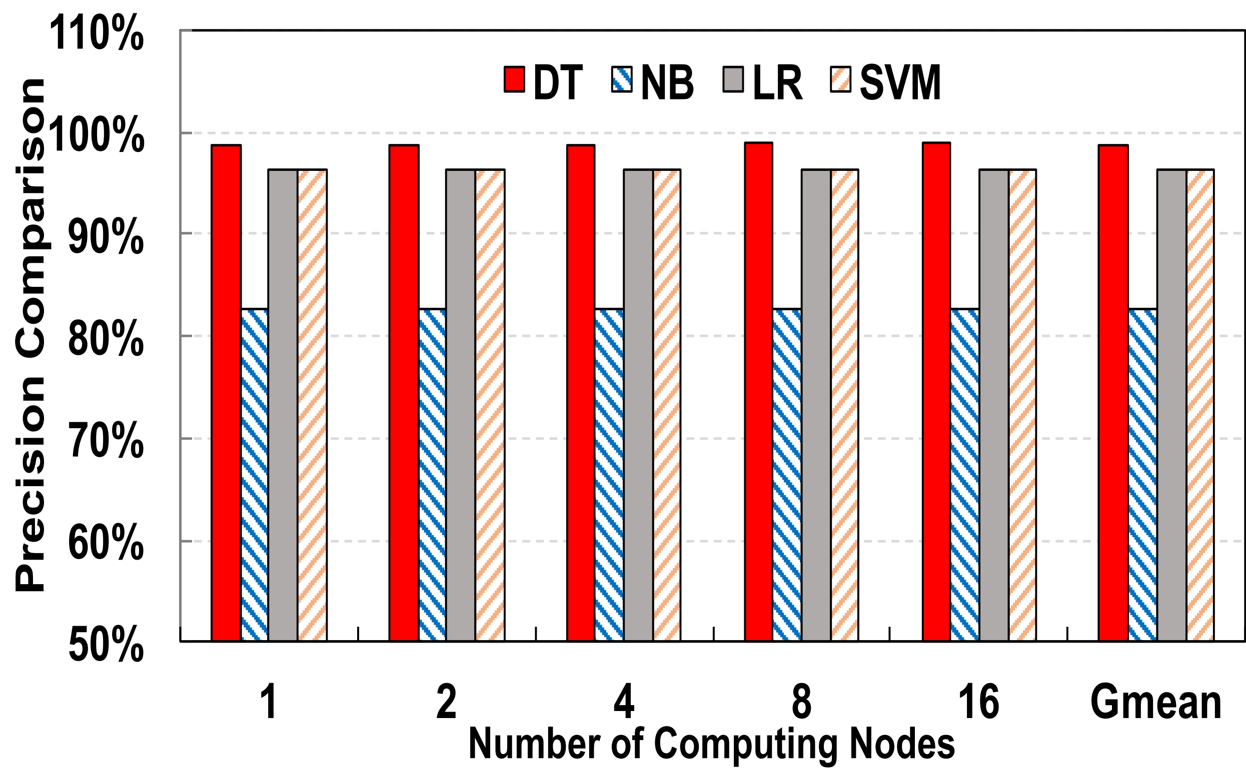

Figure 5.24: Precision Comparison of DT, NB, LR, and SVM Classifiers

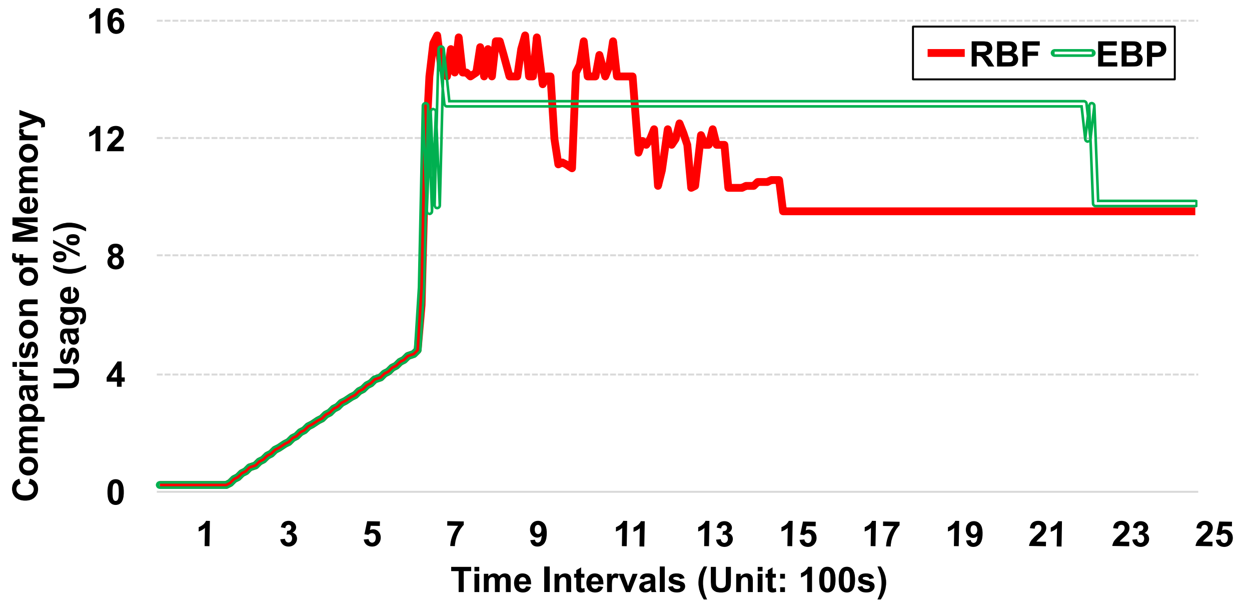

Figure 5.25: Memory Usage of RBF and EBP

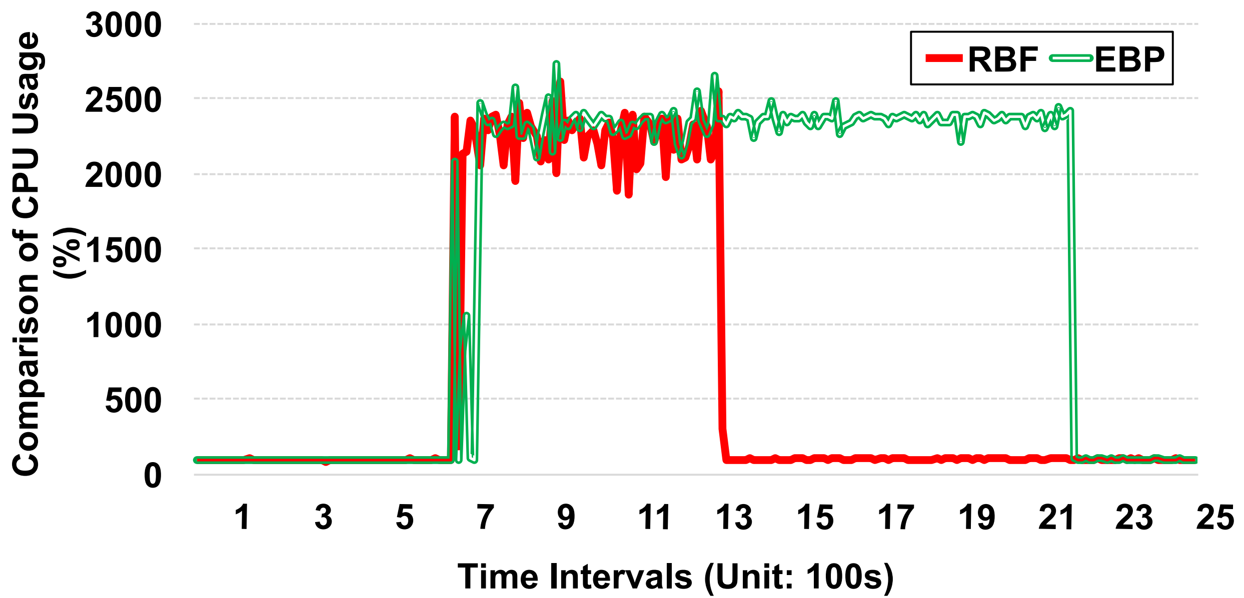

Figure 5.26: CPU Usage of RBF and EBP

5.4 Evaluation of RBF

5.4.1 Resource Allocation Results

Fig. 5.25 shows the comparison of memory usage of RBF and EBP networks. We can see that the EBP network uses 2200 s to finish the training job and the RBF network finishes its

job with less time (1400 s). In details, RBF and EBP networks request the similar memory resources to load the whole dataset to the memory from the disk in 700 s due to the same data sizes. The RBF network continues to calculate the cluster centers of the dataset with larger memory resources than the EBP network between 700 s and 1100 s. When the RBF network finishes the computation of the data centroids, its memory usage decreases by 6% approximately and its training procedure is performed from 1100 s to 1500 s. In contrast, the EBP network requires less memory than the RBF network during the process of cluster centers' calculation from 700 s to 1100 s. Then its need of memory exceeds the RBF network before accomplishing the training procedure.

Fig. 5.26 shows the comparison of their CPU usages. The EBP network spends less execution time (1300 s) to train the classification model than the EBP network (2100 s). At 700 s, RBF network and EBP network load the whole dataset to the memory for the numerical computation. When finishing the loading job, the RBF network and the EBP network begin to train the model from 700 s to 2200 s. We can see that the RBF network calculates the neuron information with lower CPU usage (1960%) than the EBP network (2160%).Because we implement and execute the two methods on multiple CPU cores, the percentage of their CPU usage can reach more than 100%. When the RBF network accomplishes the computation at 1300 s, its CPU usage decreases to 100% dramatically, which is similar with the EBP network at 2100 s.

5.4.2 Comparison of Accuracy Rate

Fig. 5.27 shows the accuracy rates of the RBF network and the EBP network. We can see that the RBF network can lead to a higher accuracy rate (93.64%) than the EBP network (83.57%) on average. In the experiment, 1275 malware and 1275 benign applications are used to train the precise model. In contrast to the EBP network which iteratively reduces the error for all data samples, the RBF network can preserve 96% of the accuracy rate since its clustering centers aggregate the similar application samples into same regions.

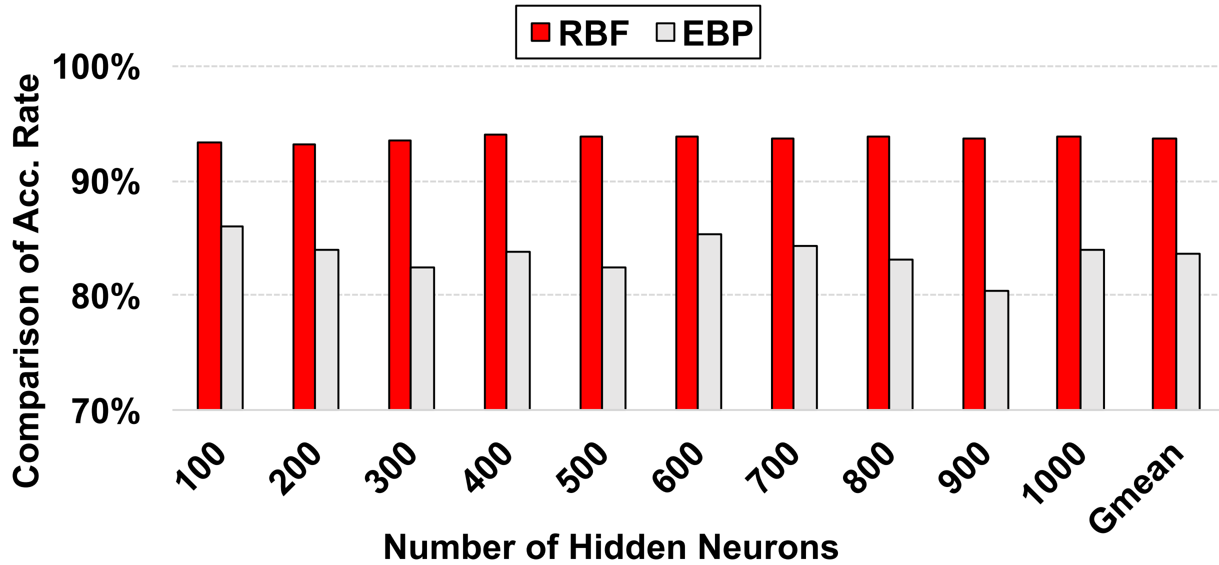

Figure 5.27: Accuracy Rate of RBF and EBP with Hidden Neurons

The RBF network chooses the clustering centers as its hidden neurons, however, the EBP network retrieves the hidden neurons through the iterative computation of the entire dataset. Additionally, the training results of the EBP network alters from 80% to 85% along with the changes of the number of hidden neurons. However, the RBF network can avoid the situation due to its clustering computation and gradient descent method.

Chapter 6

Summary and Future Work

In our study, we analyzed the performance issues for selecting relevant features that are effective for detecting malicious apps on the Android platform. Accordingly, we designed a multiple dimensional kernel feature-based malware detection infrastructure and implemented a multiple dimensional kernel feature's collection agent to dynamically collect, transfer, and store our 112-dimension data. We have examined 275 malware apps each of which has 15,000 instances and 275 benign apps with the same number of instances. The effective dimensional reduction algorithms, PCA, Correlation, Chi-square and Info Gain, are also employed to dig out the more important features to malware detection. By using more signal- and memory-related features of Android kernel, classifiers of Naive Bayes, Decision Tree and Neural Network efficiently achieve 94%-98% of accuracy rate and less than 10% of false positive rate. In contrast to Naive Bayes , Decision Tree and Neural Network can predict more precisely the malicious apps while avoid the issue of overfitting. These results demonstrate that characterization of kernel features is directly relevant to predicting the malware presence accurately.

Furthermore, we proposed an automatic data collector mechanism and a Spark-based malware detection framework. This data collector mechanism implements the collection, storage, and transfer of our large-scale dataset. The Spark-based malware detection architecture accurately deals with the original data sample from the data collector and efficiently predict malicious behaviors in memory. To the end, this paper demonstrates the sensitiveness of NB, DT, SVM and LR classifiers on Apache Spark platform, in which the DT classifier can preserve a higher precision rate and eliminate the execution cost. Moreover, our Spark-based malware detection technique improves its performance when the data size

dramatically increases. The time consumption is also optimized by using less frequent I/O communications.

In addition, we enhanced a RBF network based malware detection technique with a heuristic approach of clustering. To measure the similarity in Android datasets, the K-means algorithm calculates the centroids of all data samples in each cluster for initializing the hidden neurons of the RBF network, which assigns each data point from a large-scale dataset into different regions. According to the initialized hidden centers, the RBF network can quickly and precisely compute the positions for unknown data samples through the correct Gaussian functions. Finally, this dissertation demonstrates the forensics analysis of the main kernel parameters, the resource usage of the RBF network and EBP network and their performances. The RBF network can preserve a higher accuracy rate with less execution cost and time. Moreover, compared to the EBP network, the RBF network improves its performance for the exascale computation of the large-scale dataset.

For our future work, we plan to strengthen our online framework on a parallel computing platform to reduce time delay and memory cost. When the dimension of kernel features is lower than 100, the local computation can be finished in a few minutes. If increasing the dimension of kernel features, there is a need of parallel computation for in-memory classification in powerful clusters. The future work is also divided into three parts: First, data transferring from local storage to remote database demands an effective interface. Second, it is required to load massive data to in-memory parallel computing models. Third, we will investigate the linear and nonlinear algorithms to ensure the precision of malware detection.

Bibliography

[1] 112 android kernel parameters and normalized weights. https://dochub.com/xinningwang/JZ2PZ4/mlandroid.

[2] Android malicious threats. http://usa.kaspersky.com/internet-security-center.

[3] Apache cassandra. http://cassandra.apache.org.

[4] Apache mesos. http://archive.apache.org/dist/mesos/0.27.1.

[5] Hammerhead kernel. https://android.googlesource.com/device/ lge/hammerhead-kernel.

[6] Linux.trojan.ddos description. https://blog.avast.com/2015/01/06/linux-ddos-trojan-hiding-itself-with-an-embedded-rootkit/.

[7] Rapidminer. https://rapidminer.com.

[8] Scala. http://www.scala-lang.org.

[9] Trojan.arcbomb description. https://securelist.com/threats/trojan-arcbomb/.

[10] Trojan.ransom.gen description. http://www.enigmasoftware.com/trojan-ransom-gen-removal/.

[11] Trojan.rootkit description. http://www.enigmasoftware.com/trojan-rootkit-gen-variants-block-security-applications/.

[12] Trojan.spy description. https://www.f-secure.com/v-descs/pswsteal.shtml.

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

[14] B. Amos, H. Turner, and J. White. Applying machine learning classifiers to dynamic android malware detection at scale. In *Wireless communications and mobile computing conference (iwcmc), 2013 9th international*, pages 1666–1671. IEEE, 2013.

[15] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[16] L. Armijo. Minimization of functions having lipschitz continuous first partial derivatives. *Pacific Journal of mathematics*, 16(1):1–3, 1966.

[17] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

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

[19] L. Batyuk, M. Herpich, S. A. Camtepe, K. Raddatz, A.-D. Schmidt, and S. Albayrak. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications. In *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*, pages 66–72. IEEE, 2011.

[20] M. Y. Becker and P. Sewell. Cassandra: Flexible trust management, applied to electronic health records. In *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*, pages 139–154. IEEE, 2004.

[21] C. M. Bishop. *Neural networks for pattern recognition.* Oxford university press, 1995.

[22] C. M. Bishop. *Pattern recognition and machine learning.* springer, 2006.

[23] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak. An android application sandbox system for suspicious software detection. In *Malicious and unwanted software (MALWARE), 2010 5th international conference on*, pages 55–62. IEEE, 2010.

[24] D. Borthakur. Hdfs architecture guide. *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf*, 2008.

[25] A. P. Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern recognition*, 30(7):1145–1159, 1997.

[26] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. *Technische Universität Darmstadt, Technical Report TR-2011-04*, 2011.

[27] 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.

[28] P. P. Chan, L. C. Hui, and S.-M. Yiu. Droidchecker: analyzing android applications for capability leak. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 125–136. ACM, 2012.

[29] S. Chen, C. F. Cowan, and P. M. Grant. Orthogonal least squares learning algorithm for radial basis function networks. *IEEE Transactions on neural networks*, 2(2):302–309, 1991.

[30] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 239–252. ACM, 2011.

[31] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *Proceedings of the 1st India software engineering conference*, pages 5–14. ACM, 2008.

[32] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu. Large-scale malware classification using random projections and neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3422–3426. IEEE, 2013.

[33] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[34] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo. On the feasibility of online malware detection with performance counters. *ACM SIGARCH Computer Architecture News*, 41(3):559–570, 2013.

[35] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra. Madam: a multi-level anomaly detector for android malware. In *International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*, pages 240–253. Springer, 2012.

[36] S. Dreiseitl and L. Ohno-Machado. Logistic regression and artificial neural network classification models: a methodology review. *Journal of biomedical informatics*, 35(5):352–359, 2002.

[37] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern classification*. John Wiley & Sons, 2012.

[38] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan. Android security: a survey of issues, malware penetration, and defenses. *Communications Surveys & Tutorials, IEEE*, 17(2):998–1022, 2015.

[39] T. Fawcett. An introduction to roc analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

[40] I. K. Fodor. A survey of dimension reduction techniques, 2002.

[41] D. François. High-dimensional data analysis. *From Optimal Metric to Feature Selection, VDM Verlag, Saarbrucken, Germany*, pages 54–55, 2008.

[42] M. Garnaeva, J. Wiel, D. Makrushin, and Y. N. Anton Ivanov. Kaspersky security bulletin 2015. *Spam Evolution*, 2015.

[43] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale. In *International Conference on Trust and Trustworthy Computing*, pages 291–307. Springer, 2012.

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

[45] M. T. Hagan, H. B. Demuth, M. H. Beale, and O. De Jesús. *Neural network design*, volume 20. PWS publishing company Boston, 1996.

[46] H.-S. Ham and M.-J. Choi. Analysis of android malware detection performance using machine learning classifiers. In *ICT Convergence (ICTC), 2013 International Conference on*, pages 490–495. IEEE, 2013.

[47] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[48] H. Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of educational psychology*, 24(6):417, 1933.

[49] T. Isohara, K. Takemori, and A. Kubota. Kernel-based behavior analysis for android malware detection. In *Computational Intelligence and Security (CIS), 2011 Seventh International Conference on*, pages 1011–1015. IEEE, 2011.

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

[51] J. Kim, Y. Yoon, K. Yi, J. Shin, and S. Center. Scandal: Static analyzer for detecting privacy leaks in android applications. *MoST*, 12, 2012.

[52] R. Kohavi and J. R. Quinlan. Data mining tasks and methods: Classification: decision-tree discovery. In *Handbook of data mining and knowledge discovery*, pages 267–276. Oxford University Press, Inc., 2002.

[53] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X.-y. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *USENIX security symposium*, pages 351–366, 2009.

[54] A. Kusiak. Feature transformation methods in data mining. *Electronics Packaging Manufacturing, IEEE Transactions on*, 24(3):214–221, 2001.

[55] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. Accessminer: using system-centric models for malware protection. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 399–412. ACM, 2010.

[56] E. L. Lehmann and J. P. Romano. *Testing statistical hypotheses*. Springer Science & Business Media, 2006.

[57] O. Linda, T. Vollmer, and M. Manic. Neural network based intrusion detection system for critical infrastructures. In *2009 international joint conference on neural networks*, pages 1827–1834. IEEE, 2009.

[58] L. Liu, G. Yan, X. Zhang, and S. Chen. Virusmeter: Preventing your cellphone from spies. In *Recent Advances in Intrusion Detection*, pages 244–264. Springer, 2009.

[59] R. Love, S. H. W. Are, A. C. Linus, L. V. C. U. Kernels, and B. W. Begin. *Linux Kernel Development Second Edition*. Novell Press: Sams Publishing, 2005.

[60] X. Meng, J. Bradley, B. Yuvaz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. Mllib: Machine learning in apache spark. *JMLR*, 17(34):1–7, 2016.

[61] J. Moody and C. J. Darken. Fast learning in networks of locally-tuned processing units. *Neural computation*, 1(2):281–294, 1989.

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

[63] J. R. Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

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

[65] J. Rhee, R. Riley, D. Xu, and X. Jiang. Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory. In *Recent Advances in Intrusion Detection*, pages 178–197. Springer, 2010.

[66] I. Rish. An empirical study of the naive bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41–46. IBM New York, 2001.

[67] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[68] J. Sahs and L. Khan. A machine learning approach to android malware detection. In *Intelligence and security informatics conference (eisic), 2012 european*, pages 141–147. IEEE, 2012.

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

[70] A.-D. Schmidt, J. H. Clausen, A. Camtepe, and S. Albayrak. Detecting symbian os malware through static function call analysis. In *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on*, pages 15–22. IEEE, 2009.

[71] 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.

[72] F. Schwenker, H. A. Kestler, and G. Palm. Three learning phases for radial-basis-function networks. *Neural networks*, 14(4):439–458, 2001.

[73] A. Shabtai, Y. Fledel, and Y. Elovici. Automated static code analysis for classifying android applications using machine learning. In *Computational Intelligence and Security (CIS), 2010 International Conference on*, pages 329–333. IEEE, 2010.

[74] 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.

[75] F. Shahzad, M. Akbar, S. Khan, and M. Farooq. Tstructdroid: Realtime malware detection using in-execution dynamic analysis of kernel process control blocks on android. *National University of Computer & Emerging Sciences, Islamabad, Pakistan, Tech. Rep*, 2013.

[76] F. Shahzad, M. Shahzad, and M. Farooq. In-execution dynamic malware analysis and detection by mining information in process control blocks of linux os. *Information Sciences*, 231:45–63, 2013.

[77] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. IEEE, 2010.

[78] E. Skoudis and L. Zeltser. *Malware: Fighting malicious code*. Prentice Hall Professional, 2004.

[79] S. Smalley and R. Craig. Security enhanced (se) android: Bringing flexible mac to android. In *NDSS*, volume 310, pages 20–38, 2013.

[80] N. Smyth. *Android Studio 2 Development Essentials*. eBookFrenzy, 2016.

[81] D. Stopel, Z. Boger, R. Moskovitch, Y. Shahar, and Y. Elovici. Improving worm detection with artificial neural networks through feature selection and temporal analysis techniques. *Int J Comput Sci Eng*, 15:202–208, 2006.

[82] D. J. Tan, T.-W. Chua, V. L. Thing, et al. Securing android: A survey, taxonomy, and challenges. *ACM Computing Surveys (CSUR)*, 47(4):58, 2015.

[83] J. Tao and T. Tan. Affective computing: A review. In *Affective computing and intelligent interaction*, pages 981–995. Springer, 2005.

[84] G. Tuvell, C. Jiang, and S. Bhardwaj. Off-line mms malware scanning system and method, Feb. 11 2008. US Patent App. 12/029,451.

[85] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[86] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Profiledroid: multi-layer profiling of android applications. In *Proceedings of the 18th annual international conference on Mobile computing and networking*, pages 137–148. ACM, 2012.

[87] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*, pages 62–69. IEEE, 2012.

[88] L. Xie, X. Zhang, J.-P. Seifert, and S. Zhu. pbmds: a behavior-based malware detection system for cellphone devices. In *Proceedings of the third ACM conference on Wireless network security*, pages 37–48. ACM, 2010.

[89] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[90] S. Y. Yerima, S. Sezer, G. McWilliams, and I. Muttik. A new android malware detection approach using bayesian classification. In *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*, pages 121–128. IEEE, 2013.

[91] L. Yu and H. Liu. Feature selection for high-dimensional data: A fast correlation-based filter solution. In *ICML*, volume 3, pages 856–863, 2003.

[92] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Technical report, Technical Report UCB/EECS-2011-82, EECS Department, University of California, Berkeley, 2011.

[93] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10:10–10, 2010.

[94] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Presented as part of the*, 2012.

[95] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 93–104. ACM, 2012.

[96] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In *NDSS*, volume 25, pages 50–52, 2012.