TRACING MALICIOUS ANDROID APPLICATIONS FROM MEMORY DUMPS

Khalid Imran
01-247202-008
Prof. Dr. Faisal Bashir Hussain

A thesis submitted in fulfilment of the requirements for the award of degree of Masters of Science (Information Security)

Department of Computer Science

BAHRIA UNIVERSITY ISLAMABAD

September 2022

Approval of Examination

Scholar Name: Khalid Imran
Registration Number: 71076
Enrollment: 01-247202-008
Program of Study: MS - Information Security
Thesis Title: Tracing malicious Android applications from memory dumps

It is to certify that the above scholar's thesis has been completed to my satisfaction and, to my belief, its standard is appropriate for submission for examination. I have also conducted plagiarism test of this thesis using HEC prescribed software and found similarity index 3% that is within the permissible limit set by the HEC for the MS/M.Phil degree thesis. I have also found the thesis in a format recognized by the BU for the MS/M.Phil thesis.

Principal Supervisor Name: Prof. Dr. Faisal Bashir Hussain
Principal Supervisor Signature:
Date: 30/09/2022

Author's Declaration

I, **Khalid Imran** hereby state that my MS/M.Phil thesis titled **"Tracing Malicious Android Applications from Memory Dumps"** is my own work and has not been submitted previously by me for taking any degree from Bahria University or anywhere else in the country/world. At any time if my statement is found to be incorrect even after my graduation, the University has the right to withdraw/cancel my MS/M.Phil degree.

Khalid Imran
Date: 30/09/2022

Plagiarism Undertaking

I, solemnly declare that research work presented in the thesis titled **Tracing Malicious Android Applications from Memory Dumps** is solely my research work with no significant contribution from any other person. Small contribution / help wherever taken has been duly acknowledged and that complete thesis has been written by me. I understand the zero tolerance policy of the HEC and Bahria University towards plagiarism. Therefore I as an Author of the above titled thesis declare that no portion of my thesis has been plagiarized and any material used as reference is properly referred / cited.

I undertake that if I am found guilty of any formal plagiarism in the above titled thesis even after award of MS/M.Phil degree, the university reserves the right to withdraw / revoke my MS/M.Phil degree and that HEC and the University has the right to publish my name on the HEC / University website on which names of scholars are placed who submitted plagiarized thesis.

Khalid Imran

Date: 30/09/2022

Dedication

To my great mentor Abdur Rahman (My late Grandfather), my beloved Parents, Wife and Children. Thank you for all your support along the way.

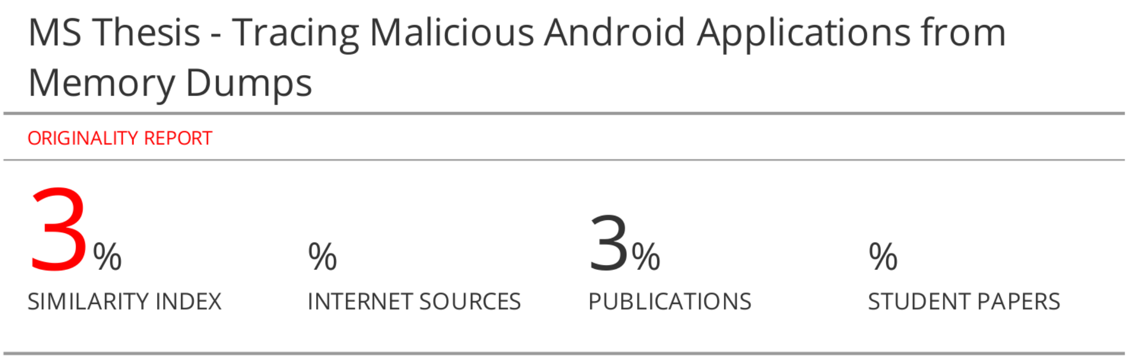
Khlaid Imran

Acknowledgements

In the name of ALLAH, the Most Gracious, the Most Merciful. First and Foremost, I am thankful to Almighty ALLAH, for giving me strength and knowledge to complete this task.

In preparing this thesis, I was in contact with many people, researchers, academicians, and practitioners. They have contributed towards my understanding and thoughts. In particular, I wish to express my sincere appreciation to my thesis supervisor, Prof. Dr. Faisal Bashir Hussain, for encouragement, guidance, critics and friendship. He has always been the enthusiastic supporter of my work. His help, made this research accomplishable. Without his continued support and interest, this thesis would not have been the same as presented here.

I am also grateful to Ms. Saneeha Khalid (PhD Scholar) for her relentless guidance and support throughout the thesis.



MS Thesis - Tracing Malicious Android Applications from Memory Dumps

ORIGINALITY REPORT

3%	%	3%	%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

Figure 1: Summery of Originality Report

Abstract

With the rapid increase in smartphone users, Android has become the most widely used Operating System in mobile devices. Due to its popularity, Android is the prime target for malicious applications, which poses a serious and evolving security threat to these devices. Existing studies focused on statistical features such as intent, permissions, API calls, and entropy for the detection of malicious apps. It is difficult to achieve a high degree of accuracy using static analysis due to the growing use of modern obfuscation techniques in Android applications. In recent years, dynamic analysis has come out as the front runner for in-depth analysis of software applications. Contemporary studies have shown efficient malware detection using resource consumption, opcode, heap dump information, object reference graph, and Process Control Block (PCB), which are extracted from process memory. Among the aforementioned feature sources, PCB contains the most in-depth and precise working information for the analysis of Android applications. Due to the complex structure of PCB in Linux-based Operating System, very limited existing work has explored the possibility of malware detection using PCB. In this study, a framework for fingerprinting malicious Android applications is presented. The implemented framework is capable of installing, executing, issuing pseudorandom events to the application, dumping memory of the device, extract PCB from memory dump and saving the result to a datastore (csv file). We extracted a comprehensive feature set that comprises of 526 features, which are then reduced to 98 features for identification and categorization of Android applications into five distinct categories. The proposed feature set is evaluated by using Decision Tree, NB, SVM and KNN machine learning classifiers. The results demonstrate that the proposed PCB-based features can significantly improve malware detection using Decision Tree and SVM.

TABLE OF CONTENTS

LIST OF TABLE

LIST OF FIGURE

Acronyms and Abbreviations

PCB	Process Control Block
ART	Android Runtime
HAL	Hardware Abstraction Layer
UI	User Interface
APK	Android Package Kit
OS	Operating System
IDE	Integrated Development Environment
CPU	Central Processing Unit
LiME	Linux Memory Extractor
LKM	Loadable Kernel Module
ADB	Android Debug Bridge
API	Application Program Interface
VM	Virtual Machine
IG	Information Gain
NB	Naive Bayes
KNN	K-Nearest Neighbors
SVM	Support Vector Machine
DNN	Deep Neural Networks
ANN	Artificial Neural Network
DT	Decision Tree

CHAPTER 1

INTRODUCTION

The mobile devices industry has evolved drastically in the recent past. According to Statista [2], around 6.2 billion smartphone users were reported in 2021, and expected that the number would reach 7.6 billion in 2027. Among the different operating systems available for mobile devices, Android is the most widely used operating system, holding a share of around 71.8% in the market of mobile devices worldwide [3]. As Android is open source and supportive towards developers, which makes it attractive for developers to develop applications for it. Currently a large number of different applications are available on Google Play Store and the number increases on daily basis. Around 89 thousand new applications are released on Google Play Store in June 2022 [4]. Besides the security checks imposed by Google Play Store, there are still a lot of malicious applications available [5]. As per statistics reported by statista [6], 10.5 million new Android malware were developed during 2019, and the development continued to grow with a rate of 0.48 million per month during 2020. The increasing number of malware poses a serious and evolving security threat to the Android-based mobile devices [7]. Therefore various schemes have been proposed by researchers to protect and depend these devices.

Security tools offered by security vendors, which are mostly based on signature based schemes are widely used for mitigating the security threat posed by malware. These tools are very efficient in detecting known malware but are unable to detect zero day or newer malware and are also less efficient in detecting malware variants [8]. Various behavioural based schemes are proposed to overcome the limitation of signature based schemes. These schemes are using generic footprint or patterns for detection and categorization of malware. Machine learning based methods can be adopted for generation of generic behavioural patterns for malware identification and categorization. However, an effective machine learning based identification and categorization model depends on the selection of useful features that are efficient in predicting the response. When it comes to Android's malware identification and categorization, there are two types of features that have been used so far; static and dynamic [1].

Static analysis is the examination of an application (code, executables, or other related files) without executing it. When it comes to Android's applications, AndroidManifest.xml and calsses.dex files are normally used to extract static features[9]. The manifest file contains information about the application components (such as activities, services, content providers, etc.), name of the class file of components, intents (which describe how the components to be activated), and permissions that the application requires during execution[10]. The .dex (Dalvik Executable) file is a compressed file containing all the class files of an application, whereas class files are the bytecode representation of the application code. The .dex file is used by many studies for static analysis of malicious code patterns[11]. Dynamic analysis refers to analysing the application behaviour, while the application is running. Dynamic analysis provides a deeper visibility of the application and hence it is obfuscation resilient[12]. The run-time behaviour of an application includes API usage[13], network communication[14], memory usage[15] and resource consumption[12].

Obfuscation is arranging the code structure in such a way that makes it difficult for reverse engineering techniques to understand. Common obfuscation strategies include junk code insertion, string encryption, class encryption, control flow manipulation, members reordering and identifier renaming[16, 17]. Static analysis schemes are most affected by obfuscation techniques. In case of class encryption, the structure of code is totally hidden from static analysis[17]. However dynamic analysis techniques and especially the memory based analysis, analyse the run-time behaviour of application, where the application code is exposed for running to perform the desired activity and is therefore more resilient to many obfuscation schemes[12, 15].

This study proposes a dynamic analysis technique based on process-specific memory-based artifacts of Android Operating System. The proposed framework captures volatile memory dump of Android based device, during execution of the target application, while the process-specific artifacts are extracted from the Process Control Block (PCB) of the target process in memory dump, which represents behaviour profile of the application. As the result or output of machine learning algorithm is highly dependent on the input data fed to the algorithm. It implies that the input data should have enough attributes for defining the output, i.e., too limited attributes will result in overfitting issue, or attributes that does not cover all the aspects will produce poor results. The feature extraction process yields a comprehensive and effective feature set, that comprises of 526 attributes, divided into 9 categories, from process control block of Android operating system. The feature-set is evaluated by employing various feature selection techniques and machine learning classifiers for identification and categorization of Android malware into five distinct categories. Moreover, to ensure effectiveness of features and add variability to the application's behaviour and memory dumps, the dumps are captured at 4 different times of an application's execution. Also, to ensure code coverage of the application, random events are generated against the application being analyzed before the memory dump is captured.

The contribution of this study are as follows.

1. We designed and implemented a software agent for acquisition of memory dumps from Android-based devices and feature extraction from the captured memory dump samples. The software agent is responsible for dataset creation containing the feature-set from process control block (`task_struct`) of Android Operating System.

2. We believe that this is the first study that uses features from the Android kernel task structure using volatile memory dumps for detection and categorization of unknown malware.

3. We created a feature-set by collecting 526 features from 10,000 memory samples extracted after execution of 2500 applications, randomly selected from CICMalDroid 2020 [18] dataset of apk files. The feature set is comprised of 425 newly identified features and 101 previously identified features, used by [1, 19] for malware detection. The remaining 11 features identified by [1, 19] does not exist in the current kernel version (4.4) which is being used in the experiment.

4. We evaluated the previously identified 112 `task_struct` features set and the improved feature set, comprising of 526 features, for identification and categorization of malware and showed their impact on the identification and categorization of Android malware.

5. We proposed a feature set comprising of 98 features, which consists of 70 newly identified features, while the rest of 28 features are part of 112 features identified by [1].

1.1 Problem Motivation

Android-based smartphones are the prime target of malicious applications due to their popularity in the smartphone industry. Malicious applications not only affect the performance of the phone by utilizing its resources but also pose security threats to user privacy by stealing personal or confidential data. To protect against these threats, security vendors offer tools to identify and classify malicious applications. These tools normally uses some signature based techniques for identification of malicious applications. These signatures need to be updated frequently in order to identify new threats, i.e., new malware needs to be analysed by the security analyst and the database of the security tool needs to be updated with the new signatures. Behavioural based techniques, which are using behavioural patterns (static/dynamic) for detection of malicious applications, are used to overcome the limitations of signature based schemes. Among the behavioural based analysis techniques, dynamic analysis is the leading candidate because static analysis schemes are badly affected by modern code obfuscation techniques, where the true picture is hidden from static analysis.

1.2 Problem Statement

In order to protect the Android information systems from malicious applications, it is eminent to use dynamic analysis techniques for the detection of application behaviour precisely. Existing solutions have shown efficient malware detection using process specific information like resource consumption [20], runtime opcode [21], heap dump information [22], object reference graph [23], extracted from process memory. Some recent studies [1, 24] have shown the strength of using Android PCB features for binary classification (malicious/benign) of Android applications. However, extensive study of Android PCB can reveal useful features, not only for binary classification but also for category classification. Category detection is a significant contribution toward malware analysis, as it not only detects the threat but also classifies the nature of the threat for taking appropriate countermeasures.

1.3 Objectives

The aim of this study is to analyse the run-time behaviour of Android applications (benign/malicious) from memory dumps for their identification and categorization. The following objectives will lead us to achieve the desired aim.

(a) Categorization of Android malware into distinct classes (Adware, Banking malware, SMS malware, Riskware, and Benign).

(b) Extraction of memory dumps after executing the application.

(c) Extraction of PCB-based features and selection of useful and effective features.

(d) Automating detection and classification of Android malware using machine learning classifiers.

1.4 Research Question(s)

In order to achieve the research objectives, following questions needs to be addressed.

(a) How to identify malware and categorize them into distinct classes using PCB-based features?

(b) How to extract memory dump after executing the application?

(c) How to extract PCB-based features from memory dumps?

(d) How to select useful and effective features that contribute to malware identification and categorization?

1.5 Significance of the Study

Antivirus software offered by security vendors are widely used to defend the information systems against the growing malware. However these software which are normally using signature based techniques for identification and classification of malware, needs to be updated with new signatures in order to identify new malware. The new signatures are created by malware analysts after a thorough investigation and analysis. This study proposes a dynamic analysis based framework for extraction of extended process specific artifacts from memory for identification and classification of malicious Android applications. The proposed framework with extended feature set will help the malware analyst in identification and classification of malware and hence strengthen the security solutions. Also, the proposed feature set can be used for detection and categorization of Android malware using dynamic analysis techniques to overcome the limitations faced by static analysis techniques due to the growing use of modern obfuscation techniques.

1.6 Thesis Organization

Rest of the document is structured as follows. The next chapter discusses background information, chapter 3 provides a detailed literature review, chapter 4 introduces the detailed research methodology, chapter 5 presents the analysis and results of the research. Conclusion and future research goals are discussed in the last chapter.

CHAPTER 2

BACKGROUND

As this study focuses on identification and categorization of Android malware. This chapter provides an overview of Android Operating System and Process Control Block (PCB), which will help in understanding the proposed framework and the feature set presented in this study. This chapter is divided into two parts, the subsequent section provides an overview of the Android architecture followed by a brief overview of the Android PCB (`task_struct`) and finally the malware analysis techniques are discussed.

2.1 Overview of Android Operating System

Android is a Linux-based open source operating system for mobile devices. It is started by Android, Inc. in 2003 and later on acquired by Google in 2005. In 2007 it is announced as an open platform for mobile devices [25]. Android is the most used operating system in the mobile industry [26]. Around 2 billion Android users were reported worldwide in 2021 [27] .As shown in figure 2.1 Android operating system consists of five layers, which can be divided into six components.

The first layer, or bottom layer contains a modified version of Linux kernel, with modifications related to embedded devices like wake locks or power manager service, support for Binder IPC, and some changes to memory management for preserving memory such as low memory killer. The Android kernel has drivers to support different hardware like camera, display, audio, WiFi, Bluetooth, keypad, etc. Besides providing support for hardware, the kernel is also responsible for providing functionality like memory management, process scheduling, different file system support, etc.[28, 29]

The next is HAL (Hardware Abstraction Layer), which exposes standard interfaces for communication with the underlying hardware to the high level API framework. These interfaces allows to implement/change functionalities without modifying the higher level components. HAL implementation is

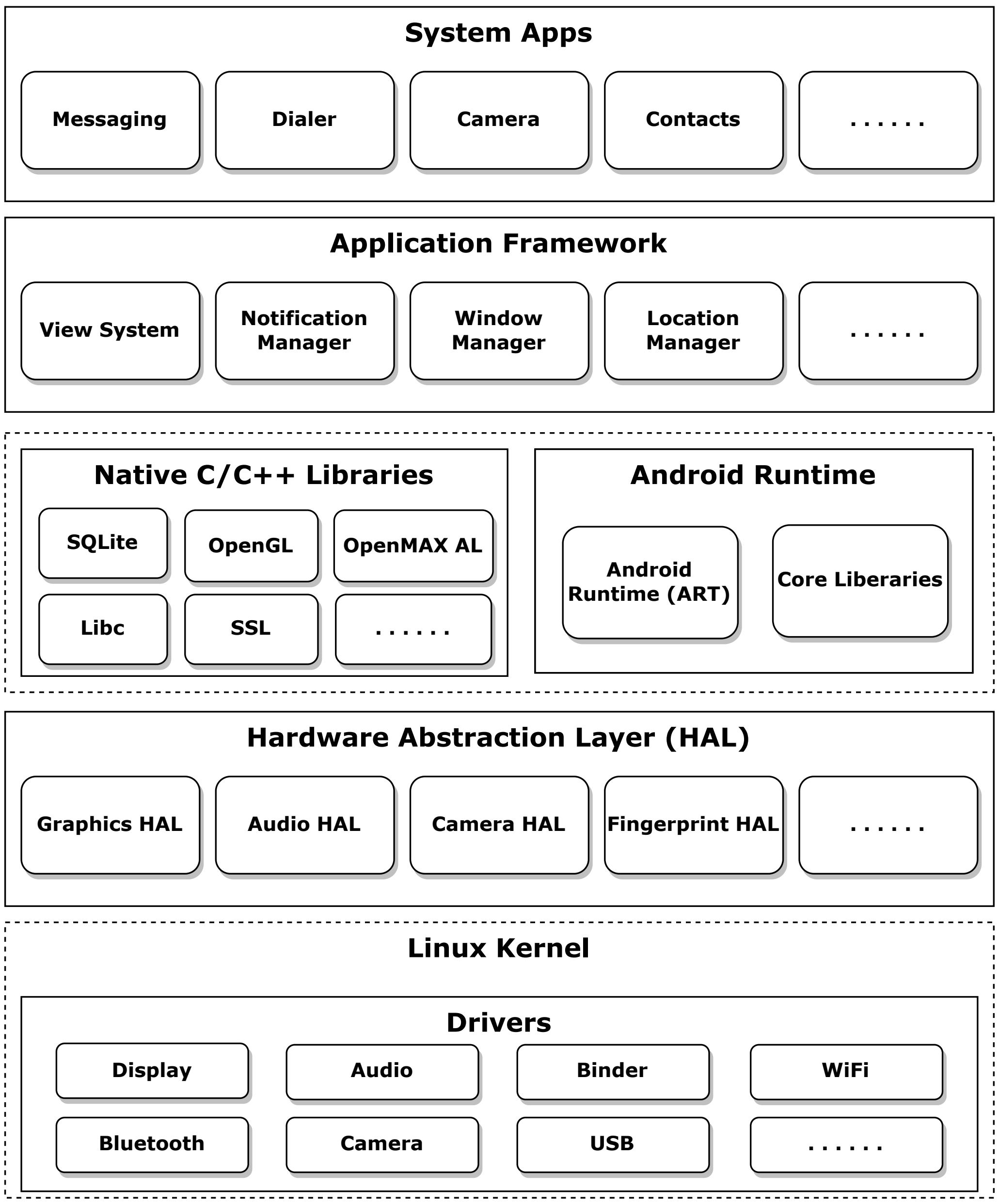


Figure 2.1: Android architecture

provided through shared libraries. when the high level API framework needs to communicate with the specific hardware, the corresponding library for the specific hardware is loaded by the Android system[30].

The next layer, which comes after HAL, consists of two components; an Android Runtime (ART) and Native C/C++ libraries. ART is a runtime environment specially written for Android based devices to execute individual processes in a sandbox or virtual environment. ART has replaced Dalvik process virtual machine and executes Dex or Dalvik executable. Native C/C++ libraries are used by some system components and services built from native code. Android NDK (a toolset for implementing applications in native code such as C/C++) is used to develop application (or part of application) to access these libraries directly from the application code.[29, 30]

The fourth layer, which is called the application framework layer, provides interfacing to all the functionalities offered by Android system to the Android apps. Android developers have full access to these APIs, to reuse the components and services offered by the Android system, such as View system (for building application's UI), Notification Manager (for displaying custom alerts), Activity Manager (for application lifecycle management), etc.[29, 30].

The last layer, which is consist of System Apps, providing the core applications such as keypad, messaging, dialer, camera etc. These apps can serve as user apps and also can be used by application developers to use their functionalities. For example if an application needs to send messaging, the developer needs not to develop the messaging capability into their application but instead will reuse the functionalities provided by the messaging app[29, 30].

2.2 Overview of Android Process Control Block (`task_struct`)

As Android operating system is based on modified Linux kernel, the Process Control Block (PCB), which is also known as the kernel task structure in Linux based systems, is a data structure of type `task_struct` located in the kernel space and includes all the information of a process; like process identifier, process name, its parent process, its siblings, thread information, CPU context, memory descriptor, scheduling information, current state of the process, file descriptor of open files, etc. The information contained in the PCB is used by the kernel for managing the scheduling and other activities of the process. Information in the process control block is effective in detection and categorization of malware because it describes the corresponding application in running state, where all its working is exposed for performing the desired activity, and hence resilient to obfuscation, where intent of the process is hidden from static analysis.

Information extracted from the process control block of Android kernel by examining the memory dump are grouped into 9 categories and are given below.

1. **Task State Information:** This category includes information like exit state, exit code, exit signal, parent process death signal etc. List of attributes in this category are presented in Table A.1

2. **Memory Information:** It describes information related to the memory held by the process and other related information required by the kernel to manage the process memory. This category comprises information like page faults, memory limits, heap address, address of code segment and data segment, information related to memory pages, pointer to executable file etc. List of attributes in this category is presented in Table A.2

3. **Signal Information:** It includes different information related to signals; their sources, handler, timer etc. List of attributes related to Signal information are described in Table A.3

4. **Scheduling Information:** The scheduling information of process includes it's scheduling state (like running, interruptible, stopped, etc), time spent while running, priority, etc. List of attributes related to scheduling information are described in Table A.4

5. **Process Credentials:** Process credentials are indicated by `cred` and `real_cred` structures in the `task_struct` and includes information related to the process security context like ownership and capabilities. List of attributes that belongs to process credentials are presented in Table A.5

6. **I/O Statistics:** I/O statistics of process is indicated by `delays` and `ioac` structures in the `task_struct` and includes information related to block I/O delay and other I/O statistical information related to the process like amount of byte read and written, the number of read and write system calls etc. List of attributes that belongs to I/O statistics are presented in Table A.6

7. **Open File Descriptors:** Open file descriptors are maintained by `files` structure in the `task_struct` and includes information related to the files opened by the process. List of attributes that belongs to open file descriptors are presented in Table A.7

8. **CPU Specific State:** The CPU specific state is maintained by the `thread` structure in the `task_struct`, and is used to save the hardware context (like register states, processor state etc) during context switch. List of attributes that belongs to CPU specific state are presented in Table A.8

9. **Others:** This category indicate miscellaneous information of the corresponding process like execution domain of self and parent process, process age, tracer flag etc. List of attributes that belongs to this category are presented in Table A.9

2.3 Malware Analysis Techniques

Malware analysis is the process of comprehending malware behaviour with the intention to mitigate the threat posed by these malware and prevent further infections [31]. Security solutions provided by security vendors are frequently employed to mitigate the evolving security threat posed by the malicious applications. These tools are typically based on signature-based methods and are quite effective at detecting known malware but fails at detecting zero-day malware and malware variants[8]. The limitation of signature-based methods is addressed by various behavioural-based methods, which are using generic footprints or behavioural patterns for fingerprinting malware. As shown in Figure 2.2, malware detection and analysis methods can be categorized into two types; static and dynamic.

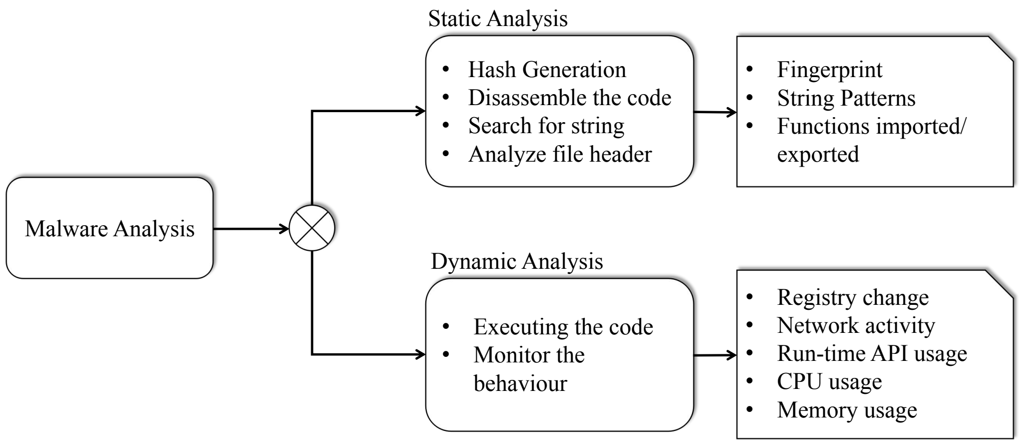


Figure 2.2: Malware Analysis Techniques

2.3.1 Static Analysis

Static analysis is the examination of application (code, executables, or other related files) without executing it. Calsses.dex and AndroidManifest.xml files are typically used to extract static features from Android applications[9]. The manifest file contains information about the program's components, name of the component's class files, intents, and permissions needed for the application to run[10]. The .dex (Dalvik Executable) file, is a compressed file that contains all of the class files of an application[11]. Although static analysis is effective in examining all potential execution paths. However, the use of modern obfuscation techniques seriously affects static analysis. Specially class encryption or dynamic code loading entirely conceal the code from static analysis[17].

2.3.2 Dynamic Analysis

Dynamic analysis is the examination of application behaviour, while the application is running[12]. The examination can be observed at various levels, it can be the run-time opcode[21], API usage[13], network communication[14], memory usage[15] or resource consumption[12]. Since dynamic analysis examines the application at run-time, where the application code is exposed for execution, to perform the intended task and is therefore more resilient to many obfuscation schemes[12, 15]. Although this technique is more resilient to many obfuscation techniques where the true picture is hidden from static analysis, but it requires a controlled execution environment to run and observe the behaviour of application.

CHAPTER 3

LITERATURE REVIEW

With the advancements in securing information systems, considerable studies have focused on the detection and categorization of unknown malware. This chapter presents research studies that have used information from the process control block (`task_struct`) of the Android OS and other Linux based systems for identification/categorization of unknown malware. This chapter also describes research studies that have used memory-based features for malware identification/categorization targeting Android and other Linux-based systems.

This chapter is divided into two parts. The first subsection describes the relevant studies using process-specific information as feature set. The second subsection summarizes the relevant studies that utilize forensic analysis of memory dumps for feature extraction, while the third subsection summarizes other relevant studies that used dynamic analysis for malware detection and categorization.

3.1 Malware detection/categorization in Android OS using dynamic analysis

Various studies have been conducted for identification/categorization of Android malware. Among those studies, the studies that used feature set from memory dumps or kernel data structure `task_struct` directly or indirectly for identification/categorization of unknown Android malware are briefly reviewed here in the following sections.

3.1.1 By using information from Process Control Block

Wang and Li [1] presented a malware detection framework for the Android platform by utilizing information from kernel data structure `task_struct`. They extracted 112 features, grouped into 5 categories from `task_struct`, against 2550 apk samples (comprising of 1275 benign and 1275 malware apk samples). Because their feature extraction method extracts 15,000 records per application per 20 seconds, therefore they used Information Gain, Correlation, Component Analysis and chi-squared statistic for lowering dimensionality of the features. A total of 70 out of 112 features are prioritized using dimensional reduction for maximizing the classification performance. The proposed framework is evaluated by using Decision Tree, K-Nearest Neighbors, Naive Bayes, and Neural Network classifiers. Their proposed framework achieved an accuracy of 94% to 98% with a false positive rate of 2% to 7%.

Kim and Choi [32] proposed features for malware detection on Android platform version 4.0 or higher. They extracted 59 features grouped into 3 classes; CPU (10 x features), memory (24 x features) and network (25 x features), from the proc filesystem of the Android platform periodically every 10 seconds against each application being examined. The proc filesystem in Android and other Linux-based systems provides interfacing between kernel structures and userspace. A total of 36 out of 59 features are selected for malware identification using Support Vector Machine (SVM) classifier for evaluating the performance. The SVM classifier resulted into an accuracy of 98.85%, TPR of 95.97% and FPR of 0.67%. The feature selection is validated by comparing results before and after feature selection. After feature selection, improvement in performance is observed, however the detection accuracy was relatively high with full feature list, but the difference was not significant.

Alawneh et al.[24] presented a malware detection framework for identification of trojanized malware. The dataset used in the experiment is comprising of 2400 APKs (1200 benign and 1200 trojanized malware). In the study, 112 features grouped into 5 categories namely task state, Signal information, CPU scheduling information Memory management information, and others, are extracted from the kernel process control block (`task_struct`). After eliminating features with zero variance, 77 features were selected out of 112 features, and finally 43 features were used to train the machine learning model. The PCB fields are recorded for 15 seconds against each APK and then sent over to a UDP server for further analysis. The proposed framework was evaluated by using BPNN (Back Propagation Neural Network), which resulted into 96.8% accuracy rate, 98% sensitivity rate, and 4.5% false alarm rate while using a PCB sequence size of 100 PCBs and the feature set comprising of 43 features. The model takes around 30 seconds for training and 73 μs for malware identification, while the information mining takes around 100 ms.

Shahzad et al. [33] presented TstructDroid, which is a real-time malware detection framework for Android-based devices. The feature set of TstructDroid is based on information extracted from the Android process control block (`task_struct`), which is comprised of 32 features selected from 99 preliminary `task_struct` fields. The feature set is created by using 110 benign and 110 malicious applications. Feature selection is carried out by eliminating the indexer fields and then using time series features shortlisting techniques (time-series difference, mean, and variance) to further reduce the feature set size. After feature extraction, the redundant instances are removed and the time series data are segmented into blocks/windows. For each block, the frequency information are extracted using Discrete Cosine Transform. Hidden patterns in process execution are identified by using the cumulative variance of features. The proposed framework is evaluated by using J48 machine learning classifier. In real-time scenario where one application is considered as testing data and all the other applications as training data, the proposed framework resulted into an accuracy of above 98% and less than 1% false alarm rate. However, by using 10-fold cross validation strategy, which is a standard methodology for evaluating machine learning classifiers, the proposed framework resulted into 90-93.6% detection rate and 5.4% to 7.3% false alarm rate.

Massarelli et al. [20] introduced AndroDFA, a dynamic malware classification technique which is based on extracting resource consumption metrics from the proc file system of Android device. 26 metrics/features grouped into 3 categories; CPU, memory, and network usage, are extracted from proc file system for fingerprint generation. Features are extracted based on DFA (Detrended Fluctuation Analysis) and pearson's correlation. Drebin and AMD dataset of malware are used in the experiment for evaluating AndroDFA. SVM classifier is used for evaluating the result, and it is showed that the proposed methodology achieves 78% and 82% accuracy with the AMD and Drebin dataset respectively.

3.1.2 Using memory-based information

Bellizzi et al. [34] proposed a framework, JIT-MF, for collecting evidence of stealthy Android attacks that uses the functionality of benign application. Functionality of the framework includes capturing process-specific memory dump at specific trigger points for the identification of in-memory data objects of benign applications which can be misused by an attacker. Identification of data objects is performed by using the Android Runtime (ART) Garbage Collector. To capture memory dump of process, repackaging the benign app with some instrumentation and re-installation is required in order to use the ART memory dump capability. Where the trigger point is selected based on heuristics. The study is evaluated by using 4 different message hijacking scenarios, launched using Metasploits Meterpreter for Android. Pushbullet, SMSonPC and Telegram are used as the target benign applications.

Zhang et al. [23] used object reference graph birthmarks (ORGB) for detecting malware in the Android operating system. The proposed system, DAMBA, uses client-server architecture for detecting malware by means of static and dynamic analysis. The client part is installed on the Android devices and is responsible for extracting the heap dump of the application and analyze it to generate object reference graph (ORG file, serve as dynamic feature for malware detection) it also extract package information including permission list and key system class list, while the server part residing remotely establish the birthmark base and decide whether the application is malicious or benign. The server part is responsible for generation of ORGB file from ORG file and also establishes the static feature base by utilizing the permission list and key system class information of the package being analyzed. Once the application is analyzed by the server, the client is updated with the result. The framework is evaluated by using 2239 malicious samples (including 1139 from the Genome Project and 1100 from Contagion) and 1000 benign apps from the Baidu App Store. In the experiment, it is showed that the proposed DAMBA framework achieved better result (Accuracy = 0.9690, Recall = 0.9855, Precision = 0.9568 and F-measure = 0.9709) than McAfee (Accuracy = 0.9381, Recall = 0.9755, Precision = 0.9124 and F-measure = 0.9429).

Benz et al. [22] presented a methodology for improving Android taint analysis. In the study, static Android taint analysis is extended with information from heap dump and also the impact of heap dump on soundness and precision is presented. The study is evaluated by modifying FlowDroid (a static taint analysis tool for Android applications) in order to integrate heap dump information in the taint analysis. FlowDroid is modified in such a way that it can capture multiple heap dumps throughout the runtime of application being analyzed, however the evaluation is performed using "separate-heaps" (information are restricted to, that are present in a single dump), "merged-heaps" (information of all the available heap dumps is considered for the analysis) and "static-fallback" (heap dumps and abstract heaps are used together) approaches. In the experiment it is showed that precision is increased while using heap dumps as an upper bound but recall is dropped by at least 77.2%. However, when heap dump is used with static-fallback approach, a comparatively better result is achieved. It is also showed that using multiple heap dumps results in better recall value, but single dump results in better runtime performance and high precision.

3.2 Malware detection in Linux-based OS by using memoroy based or Process Control Block (`task_struct`) based information

Shahzad et al. [35] proposed a dynamic analysis scheme for malware detection by using information from the process control block (PCB) of process in Linux operating system. The scheme extracted 118 parameters from the `task_struct` of Linux OS, however 16 of them are shortlisted for malware detection. Parameters shortlisting is performed by initially eliminating the parameters that do not contribute in malware detection and then time series analysis is performed for identification of parameters that have a significant contribution in malware detection. The parameters are extracted from the `task_struct` by using a customized system call framework, which mines information of the specific process every millisecond for 15 seconds. The study is evaluated by using a dataset comprising of 114 malware and 105 benign samples. It is showed that the proposed scheme achieves a detection accuracy = 96% and false alarm rate = 0% by using propositional rule learner (J-Rip).

Panker and Nissim [36] presented a framework for detection of malicious software in Linux cloud environment by capturing memory dumps from the guest operating system running in a VM. Two types of servers are used in the experimental setup; an HTTP server and a DNS server. A total of 50 malware samples and 3 fileless attacks from 9 different categories, as well as 54 benign applications and two other samples indicating the VM state (one is clean VM sample when it does not include any DNS server or HTTP server applications and the other is when the VM is running a DNS or HTTP server) are used in the study. The memory dump is captured by querying the hypervisor, while feature extraction (171 total features) from the memory dump is performed by using 21 different volatility plugins. Out of 171 features, 17 features are selected for the HTTP server and 26 for the DNS server. Four features are the same in both cases, while the rest are different. The proposed framework is evaluated by using seven different ML classifiers (SVM, Naive Bayes, Logistic Regression, KNN, Random Forest, and ANN). It is showed that the proposed framework achieved the best result with KNN (Accuracy = 0.959, TPR = 0.934 and FPR = 0.017) and DNN (Accuracy = 0.989, TPR = 0.976 and FPR = 0) classifier for unknown malware detection in HTTP server and DNS server respectively. While in case of malware categorization in HTTP and DNS servers, the best results are achieved by DNN (Accuracy = 0.986) and RF (Accuracy = 0.981) classifier respectively.

To summarize, in order to identify malicious applications and classify them, in-memory analysis of applications needed to be carried out because code obfuscation prevents static analysis to identify the behaviour of the application.

Categorizing malware is very important to estimate the threat level that the malicious application poses and for taking appropriate countermeasures. All the studies [1, 33, 24, 32] except [20] discussed here which are using information from Android process control block `task_struct` directly or indirectly, focusing on binary categorization of malware i.e., benign and malicious category. Only [20] discussed malware classification, but their features set focuses on resource consumption of malicious processes from proc filesystem and their classification accuracy is 78% and 82%. Table 3.1 presents summary of these studies.

Furthermore, the efficiency and accuracy of machine learning models are greatly dependent on the attributes that define the feature space. A detailed analysis of the Android PCB (`task_struct`) can reveal useful features in addition to the feature set proposed by these studies. The existing studies extracted features from `task_struct` grouped into five categories at maximum. However, the PCB, which defines all the aspects of a running process, can contain more effective information that can be used to more effectively fingerprint the behaviour of the process and hence may improve the detection and classification accuracy of the machine learning classifier.

Table 3.1: Summary of studies using information from Android `task_struct` or proc filesystem

Study	**Feature Source**	**Dataset**	**# of Features**	**ML Classifiers**	**Accuracy/Reliability**	**Categorization**
Wang and Li, 2021 [1]	task_struct (PCB)	1275 malware[1], 1275 benign[2] apps	10-70 out of 112	NB, DT, NN, and KNN	Accuracy: 94% to 98%, FPR: 2-7%	No
Alawneh et al., 2019 [24]	task_struct (PCB)	1200 malware[1] , 1200 benign[2] apps	43 out of 112	Back Propagation Neural Network	Accuracy: 96.8% Sensitivity: 98% FAR: 4.5%	No
Shahzad et al., 2013 [33]	task_struct (PCB)	110 malware, 110 benign apps	32 out of 99	J48	**Real-time:** Accuracy: 98%, FAR: < 1% **Cross-validation:** Accuracy: 90-93.6%, FAR: 5.4%-7.3%	No
Massarelli et al., 2020 [20]	proc filesystem	Drebin and AMD dataset	26	SVM	Accuracy: 78% (AMD), 82% (Derbin)	Yes[3]
Kim and Choi, 2014 [32]	proc filesystem	Malware samples from Ahnlab ASEC report	36 out of 59	SVM	Accuracy: 98.85%, Precision: 96.63%, FPR: 0.67%, TPR: 95.97%	No

[1]VirusTotal

[2]Google Play Store

[3]13 Families with AMD and 23 Families with Derbin dataset

CHAPTER 4

PROPOSED METHODOLOGY

The rapid increase in malware development targeting the Android based devices is a serious and evolving security threat [7]. Depending on their nature, these malicious software can harm the target system in many ways such as steals, encrypts or deletes user's data, displays unwanted popups, or consumes system resources [37]. The security threat posed by these malware demands a robust solution for identification and categorization of these malware in order to protect target devices and take appropriate countermeasures against the threat. This study proposes a framework that uses process specific attributes from the Android process control block (`task_struct`) for identification and categorization of Android malware. This chapter covers the methodology of the underlying framework. The subsequent section discusses the working mechanism of the proposed framework, then the next section provides an overview of the implementation details followed by feature selection and evaluation techniques.

4.1 Proposed Framework For Feature Extraction and Classification

Broadly, the feature extraction process is comprised of two phases; the first phase involves acquisition of memory dump from Android-based device or emulator while the candidate application is running, and the second phase starts by analyzing the memory dump for extraction of process control block of the target process and storing the result in a csv file. While the classification process involves identifying the category of application by utilizing process control block information from the csv file. Attributes of the process control block serve as feature set for identification and categorization of malware. Algorithm 1 describes the feature extraction process. The overall process of feature extraction and classification is presented in figure 4.1, while the details are presented in the subsequent section.

Algorithm 1 Feature Extractor (Memory Dump Extractor & CSV Generator)

```
1:  procedure FEATURE_EXTRACTOR
2:      if Emulator is not running then
3:          Boot Emulator
4:      end if
5:      for each $app \in$ apk_Repository do    ▷ apk_Repository is a folder on the host system containing malicious/benign applications
6:          pkg_name ← package_name(app)
7:          Install app
                                                                    ▷ Scenario #1
8:          Execute the installed app using pkg_name
9:          app_pid ← pid_of(pkg_name)
10:         Call MEMORY_DUMP_EXTRACTOR_&_CSV_GENERATOR(app_pid)
                                                                    ▷ Scenario #2
11:         Call EVENT_GENERATOR(pkg_name, $event\_count = 150$)
12:         app_pid ← pid_of(pkg_name)
13:         Call MEMORY_DUMP_EXTRACTOR_&_CSV_GENERATOR(app_pid)
                                                                    ▷ Scenario #3
14:         Call EVENT_GENERATOR(pkg_name, $event\_count = 1500$)
15:         app_pid ← pid_of(pkg_name)
16:         Call MEMORY_DUMP_EXTRACTOR_&_CSV_GENERATOR(app_pid)
                                                                    ▷ Scenario #4
17:         Call EVENT_GENERATOR(pkg_name, $event\_count = 4000$)
18:         app_pid ← pid_of(pkg_name)
19:         Call MEMORY_DUMP_EXTRACTOR_&_CSV_GENERATOR(app_pid)
20:         Reset/Clean Envoirnment
21:     end for
22: end procedure
23: procedure MEMORY_DUMP_EXTRACTOR_&_CSV_GENERATOR(pid)
24:     Capture memory dump
25:     Extract task_struct(pid)
26:     save task_struct info to csv file
27: end procedure
28: procedure EVENT_GENERATOR(pkg_name, event_count)    ▷ where pkg_name is used to distinguish the target application. While event_count is the number of events that needs be generated against the target application
29:     Run Application
30:     Send event_count pseudorandom click events to the app
31: end procedure
```

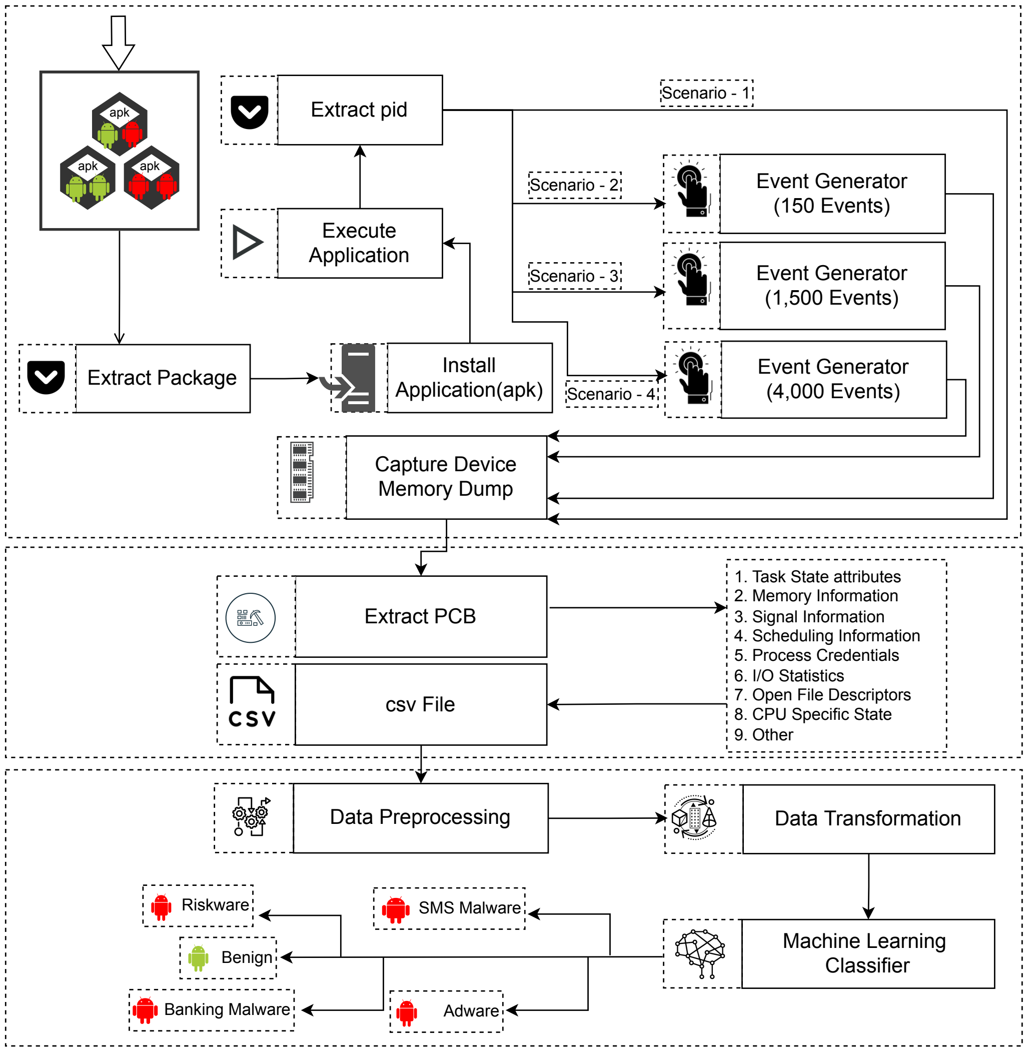


Figure 4.1: Proposed Framework for PCB based Feature Extraction and Classification of Android Malware

4.1.1 Memory Dump Acquisition Module

As shown in figure 4.1, the memory acquisition module reads the target apk and installs it into an Android-based controlled environment. Before installation, its package name is extracted, which will help in identifying its pid when the application is running. After installation, the memory acquisition module executes the application and captures volatile memory dump of the device. The acquisition of volatile memory dump is performed at four different stages; first immediately after execution, then after issuing 150 pseudo-random events to the application, then after issuing 1500 pseudo-random events to the application, and finally after issuing 4000 pseudo-random events to the application.

Although the proposed framework extracts information from the process control block of the target process, but to ensure smooth operation of the application and the device, it is necessary to consider one application at a time for installation and analysis. In order to restore the sandbox environment to clean state for the next analysis, the sandbox environment needs to be running in read-only mode to avoid persisting changes made by the installed apk. Acquisition of volatile memory dumps at different times adds variability to the dumps and thus contributes to the creation of a rich data collection which includes 10,000 volatile memory dumps. Exercising of pseudo-random events are required for ensuring code coverage and triggering of malicious behaviour.

4.1.2 PCB (Process Control Block) Extraction Module

As shown in figure 4.1, the PCB extraction module accepts the volatile memory dump file as input along with the process id of the target application. The PCB extraction module then walks through the Android PCB (`task_struct`) and extracts `task_struct` of the target application from the volatile memory dump by comparing the value of pid field in `task_struct` with the provided input value. After extraction of the target `task_struct`, it stores its attributes values into a csv file. Columns header of the csv file represents attributes of the PCB, while its row represents the values of PCB attributes from the specific volatile memory dump. The PCB extraction module extract 526 attributes from `task_struct`, which can be divided into nine categories as discussed in section 2.2. Total number of attributes against each category is described in table 4.1, while the proportion of each category in `task_struct` relative to the total extracted attributes is presented in figure 4.2.

Table 4.1: Number of Attributes Against Each Category of `task_struct`

S.No	Category	# of attributes
1	Task state Information	6
2	Memory Information	211
3	Signal Information	82
4	Scheduling Information	90
5	Process Credentials	74
6	I/O statistics	17
7	Open File Descriptors	11
8	CPU Specific State	19
9	Others	16
Total		**526**

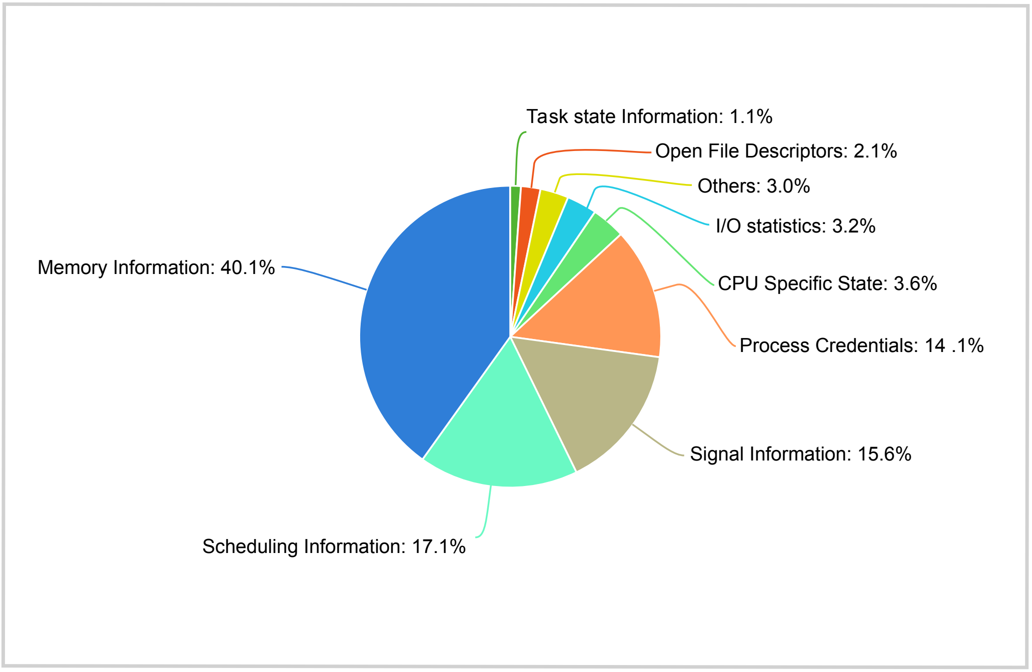


Figure 4.2: Percentage of each `task_struct` category against total extracted attributes

4.1.3 Classification Module

The classification module accepts the extracted Android PCB attributes stored in the csv file as input and produces the result by identifying and categorizing the Android application into the distinct categories. The extracted attributes from PCB serve as features for the machine learning classifier. Figure 4.3 shows the overall execution flow of the framework.

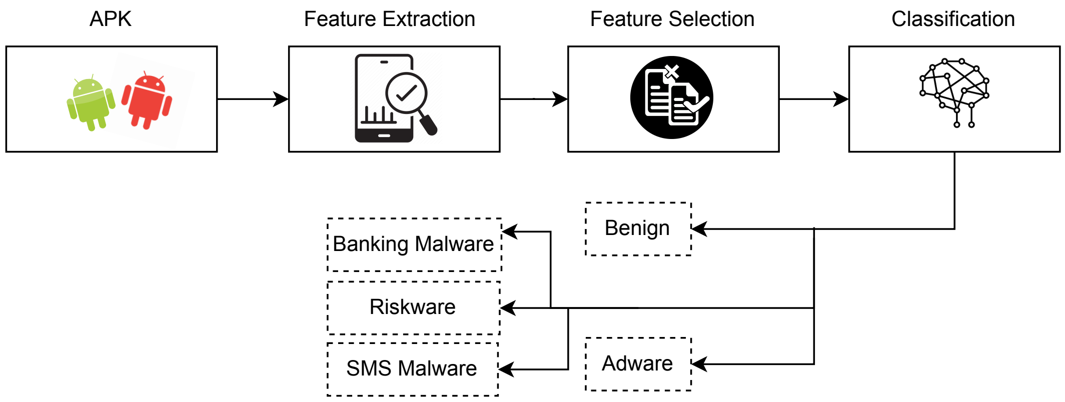


Figure 4.3: Framework Flow of Execution

4.2 Implementation of Proposed Framework

The experimental setup consists of a controlled sandboxed environment, which uses the Android Virtual Device (AVD) for defining the virtual environment. AVD helps define the virtual device by specifying its hardware profile, the amount of memory and storage allocated to the device, and system image for booting the device. AVD is hosted on a system running Ubuntu 18.04 with 10 GB of memory and 1 TB of Storage. The virtual environment is configured with Nexus 6P hardware profile, 1.5GB of RAM, 512MB SD Card, and 1GB of internal storage. The system image used by the virtual environment is Android 9.0 with Google APIs and x86_64 architecture.

The host system communicates with the virtual environment via the Android Debug Bridge (adb). Android Debug Bridge [38] is a command-line tool which helps in facilitating communication between a host system and an Android-based device by executing commands on the target device via a Unix shell provided by adb.

Monkey tool [39] which is a UI/Application exerciser, provided for stress testing of applications during development, is used as event generator for the proposed framework. Monkey tool simulates pseudo-random user input against the target application being installed during the experiment.

Android Asset Packaging Tool (aapt) [40], which is a command-line tool for viewing, creating, and updating apk files. This tool is mostly used by build scripts and IDE for packaging of Android applications. Android Asset Packaging Tool is used in the experimental setup for the extraction of package name from the target apk before their installation on the emulator. The extracted package name is used for identification of the target process among the running processes.

LiME (Linux Memory Extractor) [41], which is an open source Loadable Kernel Module (LKM) for dumping of volatile memory from Linux and Linux-based devices, over a network or to the file system of the target device, is used for acquisition of volatile memory dump from the target device. To capture memory dump, LiME is required to be compiled against the target kernel, running the Android device and loaded into it. Since LiME is LKM, so the underlying kernel needs to have enabled the loadable kernel module support. Since Android kernel does not have a default support for loadable kernel modules, so the kernel needs to be compiled with loadable module support. Goldfish kernel 4.4 is compiled and utilized in the experimental setup for booting the virtual environment.

Volatility Framework [42], which is an open source collection of tools used for analysis of volatile memory dumps captured from Windows, Mac OSX, Linux, and Linux-based devices, is used in the experimental setup for extraction of PCB from memory dump obtained through LiME from the target device. It supports different memory dump formats like Raw linear format, Hibernation files, crash dump file, LiME format, firewire etc. A Volatility profile (comprising of module.dwarf and System.map files packaged into a zip archive) is required to be built against the target kernel, for analysis of memory dump captured through LiME from the Android based device. The profile helps volatility framework in locating and parsing of information in the memory dump.

The proposed framework is implemented as a Python application that utilize the volatility APIs by importing volatility as library. The developed application has the ability to power on the virtual environment, install Android applications (apk) placed in a designated directory one by one, execute the application, simulate pseudo-random user input, capture memory dump, analyze the memory dump, extract PCB from the memory dump, and save it to a csv file. Simulation of user input is accomplished 3 times with varying numbers of inputs (150, 1500, and 4,000) and after each event generation scenario, a memory dump is taken, resulting in 4 x memory dumps against each application being analyzed. The first memory dump against a single application is taken just after execution, then after simulating 150 user inputs, then after 1,500 user inputs, and finally after issuing 4,000 user inputs.

4.2.1 Dataset and Extracted Feature-set

CICMalDroid 2020 dataset [18] of apk files is used for generation of the feature-set. The CICMalDroid 2020 dataset is comprised of 17,341 apk files divided into five categories: Benign, Riskware, Banking, SMS and Adware. The resultant features-set comprises of 526 attributes extracted from Android PCB (`task_struct`) is generated against 2500 apks files, randomly selected from CICMalDroid 2020 dataset. The 2500 randomly selected applications consist of 500 apk files from each category of the CICMalDroid 2020 dataset. Since the feature-set proposed by this study is memory-based and extracted from the Android process control block (`task_struct`), so each application is executed in a sandboxed environment and memory dump is captured. The data collection agent implemented for acquisition of memory dump and extraction of PCB attributes, captures four memory dumps against each apk file at different times. From each memory dump file, the corresponding PCB is extracted and stored into a csv file. After processing of 2500 applications, a feature-set comprising of information extracted from 10,000 memory dumps (with 2,000 memory dumps against each category of CICMalDroid 2020 dataset) is generated. The extracted feature-set from Android PCB can be divided into nine categories as described in section 2.2.

4.3 Feature Selection

The feature extraction process generates a large number of features. It is important to find the effective features as all the features in the feature set may not be significant for malware classification. Feature selection helps in reducing the complexity of model and thus reducing over-fitting [43]. It helps in identifying the features that add noise to the classification system and are a cause of low performance [44]. In order to find the set of significant features, a layered approach comprises of three layers is followed by this study. The first step analyzes all features for finding the features whose value remains same for all output classes. Such features are referred as constant features and are removed from the feature set as they have no influence on the prediction of output class. The second step focuses on non-constant features and finds Information Gain values against all features. Different thresholds for information gain are utilized for finding an optimum threshold value, which leads to a reduced feature set based on eliminating features with an IG value less than the threshold value. The resultant feature set is then passed to the next phase where features are eliminated based on their correlation, and a final set of effective features is presented. The details of all the three phases is discussed in the subsequent subsections.

4.3.1 Evaluating Constant Features for Feature Selection

The features for which the values of data samples remain same for all output classes are called Constant features. These features do not add any value to the classification system and can be regarded as ineffective features. Therefore, they should be removed from the feature set. As constant features have no variability in values, therefore the statistical measure of variance can be used for finding such features. Variance can be calculated for a feature using Eqn.4.1. Variance measures the distance of all values in a feature from the mean value of feature. If the values of a feature are diverse, variance is high and vice versa. A zero value of variance indicates that there is no variation in the values of the feature. In this study, all features are evaluated using variance and the ones with zero variance are recorded. Such features are referred as constant features.

$$s_N = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2} \tag{4.1}$$

where:

x_i is the i^{th} value of feature

$(x_i - \bar{x})^2$ is the squared distance of the i^{th} value from mean

N is the total number of samples in the data set

The remaining features will now be gauged for their significance by calculating Information Gain values against the output class.

4.3.2 Information Gain for Feature Selection

Information Gain is a measure that helps to estimate the information, a feature provides about the output class [45]. It determines the dependency between a feature and the output. If a feature is influential in determining the output, the value of Information Gain is high. Its value will be low for less influential features and zero for independent features. Information Gain can be calculated using Eqn.4.2. The values for information gain are calculated for all features selected after constant elimination.

$$I(X;Y) = H(X) - H(X|Y) \tag{4.2}$$

where:

$H(X)$ is the entropy of X (feature)

$H(X|Y)$ is the conditional entropy of X given Y

In order to find features significant for classification, a threshold value for Information Gain needs to be selected. In this study, features are selected for thresholds of 0.1, 0.2, 0.3, 0.4, 0.5, and 0.6. The threshold at which the value for classification is highest for all malware classes is chosen for the selection of final features.

4.3.3 Correlation for Feature Selection

As the name implies, correlation refers to identifying the relationship or closeness between two features. Features with high correlation are more linearly dependent, so they have the same effect on the output class. Two features having higher correlation, one of them can be dropped without much affecting the output result. In this phase, feature set is reduced by using the correlation of features.

4.4 Classification Method

After selection of the most effective features from the extracted features-set, final features-set is used for classification. Different machine learning classifiers, including Decision Tree, SVM, KNN, and Naive Bayes, are used for evaluating the feature-set. These classifiers are applied on the feature-set for identification and categorization of Android malware.

4.5 Classifier Validation Method

Cross-validation is an iterative method of evaluating the classifier based on partitioning the data into k equal parts and then applying k-1 parts for training and 1 part for testing the model iteratively. The training and testing is performed in such a manner that each part is used in testing, resulting into k tests. 10-fold cross validation is used in the study for evaluating the extracted feature set, which is the most commonly used method for evaluating the performance of machine learning classifiers. 10-fold divides the input data into 10 equal parts and evaluate the classifier performance by using 9 parts for training and 1 part for testing the classifier. The training and testing process is performed 10 times, each time with a different testing data, and the classifier performance is reported.

4.6 Classifiers Evaluation Metrics

Evaluation metrics are used to evaluate the effectiveness of machine learning classifiers for detection and classification. The subsequent subsection discusses various evaluation metrics.

4.6.1 Confusion Matrix

As the name suggests, it is a matrix that provides the complete picture of prediction model and includes the following metrics.

1. **True Positive (TP):** Predicting Yes as Yes
2. **True Negative (TN):** Predicting No as No
3. **False Positive (FP):** Predicting Yes as No
4. **False Negative (FN):** Predicting No as Yes

4.6.2 Accuracy Metric

Accuracy is the measure of prediction that a classifier correctly predicts. As shown in Eqn. 4.3, it is defined as the ratio of correct predictions to the total predictions made[46].

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{4.3}$$

4.6.3 Other Metrics

Many other metrics that are related to the confusion matrix are commonly used for evaluation of prediction models, these includes Recall, Precision, False Positive Rate, and F1-Score. These metrics can be given by equation 4.4, 4.5, 4.6, 4.7

$$Recall = \frac{TP}{TP + FN} \tag{4.4}$$

$$Precision = \frac{TP}{TP + FP} \tag{4.5}$$

$$False\ Positive\ Rate = \frac{FP}{FP + TN} \tag{4.6}$$

$$F1\ Score = 2 \times \frac{Presion \times Recall}{Precision + Recall} \tag{4.7}$$

CHAPTER 5

ANALYSIS & RESULTS

This chapter evaluates the research by presenting the experiments and results of all the applied techniques. The subsequent sections discusses feature extraction, feature selection (constant feature elimination, Information Gain and Correlation techniques in layered fashion), classification, and finally the proposed framework is evaluated by comparing the result of previously identified features with the result of improved feature-set identified by this study.

5.1 Feature Extraction

The feature extraction process works by collecting information from the process control block (`task_struct`) using volatile memory dump extracted from Android-based device. Volatile memory dump are captured 4 times from the Android based device against a single application at different times, i.e., after execution, after issuing 150, 1500, and 4000 events against the application. Each memory image is processed for collecting `task_struct` of the process under analysis. The analysis of `task_struct` leads to a rich feature set of 526 features. The large number of features is then reduced to a set of effective features by using the feature selection techniques described in the subsequent section.

5.2 Feature Selection

Since a rich set of features is extracted by the feature extraction process. A three-step feature selection process is applied for selecting effective features in a layered fashion. Initially, the features-set is reduced by eliminating the constant features, and then the resultant feature-set is passed to the next phase where Information Gain is applied. Based on experiments, an optimal value of Information Gain threshold is selected and the features whose Information Gain value is less than the selected threshold value are eliminated. Finally, the resultant feature set is passed to the next phase, where Correlation is applied and features are eliminated. The results of these three phases are presented in subsequent subsections.

5.2.1 Constant Feature Elimination

Constant features are the ones that have the same values for all output classes. This study has calculated the variance measure against all features in the data set. If the value of variance is zero, then such a feature is grouped into the set of constant features and is removed from the set of effective features. Out of 526 extracted features, 241 features are found to be constant in nature. These features are removed from the set of effective features. The remaining set of 285 features is passed to the next phase of feature selection, i.e., for calculating the information gain score.

5.2.2 Information Gain For Feature Selection

Information gain is calculated for all non-constant features selected by the previous phase. The value of Information Gain reflects the influence of the feature on prediction of output. Figures 5.1, 5.2 and 5.3 represent the IG scores for all features.

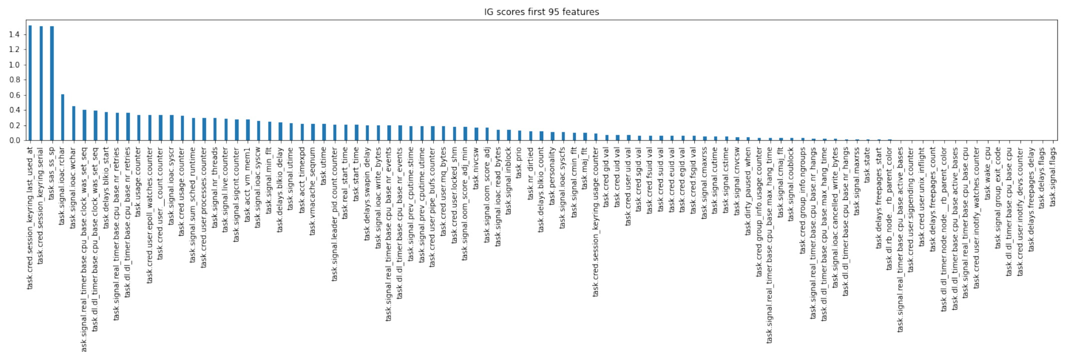


Figure 5.1: IG scores of Features (1-95)

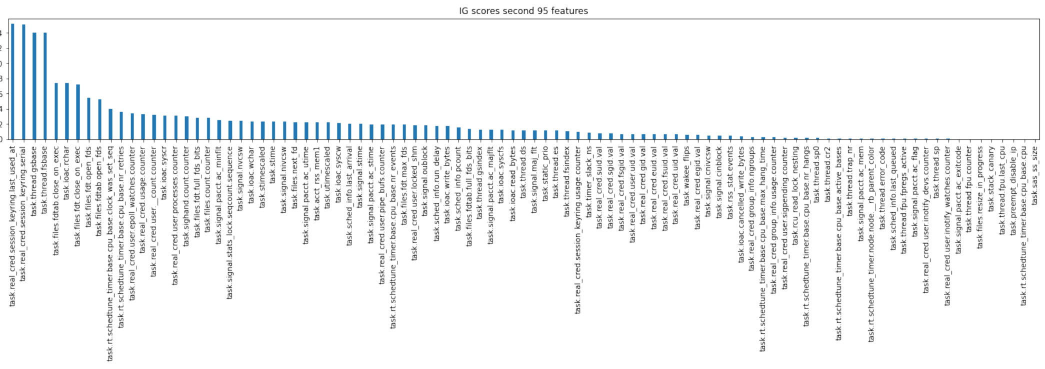


Figure 5.2: IG scores of Features (96-190)

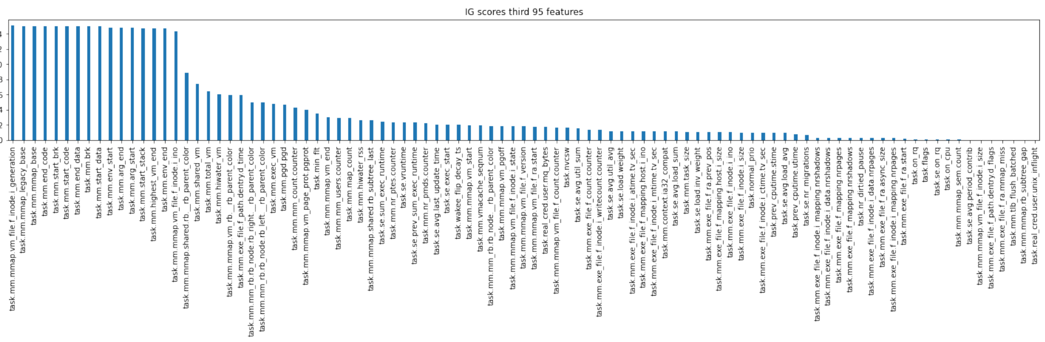


Figure 5.3: IG scores of Features (191-285)

In order to select features important for classification; a threshold estimation approach is used. The features are selected at different values of Information Gain by measuring performance measures against the selected features. The values of threshold used for feature selection are 0.1, 0.2, 0.3, 0.4, 0.5, and 0.6. Table 5.1 shows the average 10-fold cross validation accuracy measure and the resultant total number of features selected against each threshold. Figure 5.4 presents the trade-off between accuracy and IG threshold.

Table 5.1: IG Thresholds Results, Accuracy and the Resultant Features

IG Threshold	Average Accuracy	Number of Features selected
0.1	0.967	172
0.2	0.931	106
0.3	0.927	63
0.4	0.883	43
0.5	0.881	35
0.6	0.879	31

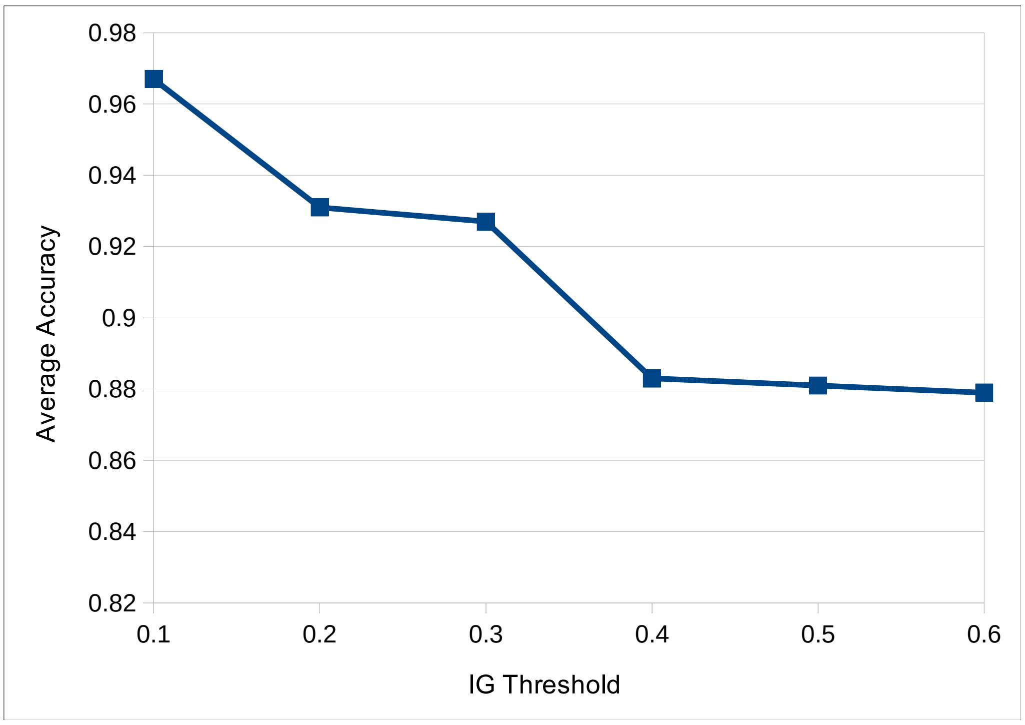


Figure 5.4: IG Threshold vs Accuracy

It can be observed that best performance is achieved at an IG threshold of 0.1 with 172 features. Therefore, these features are reported as the final selected features after phase-2 of feature selection. These features are then passed to the next phase of feature selection, which uses correlation for reducing the number of features.

5.3 Correlation

Correlation of one hundred and seventy-two (172) features, selected through the previous step of information gain, is calculated and features are eliminated based on their correlation value. Figure 5.5 shows the resultant correlation matrix. Based on the correlation, features set is reduced to ninety eight (98) from one hundred and seventy-two (172). The final feature set after applying correlation is presented in Table B.1.

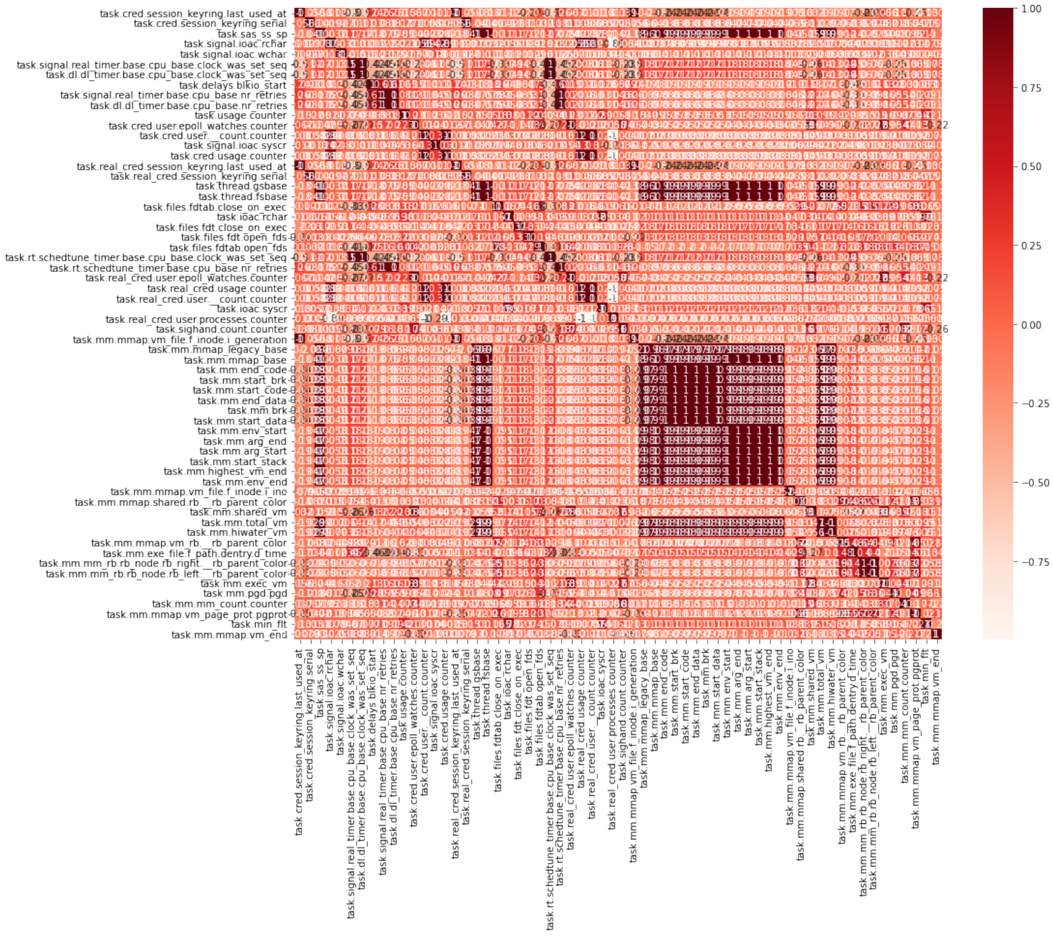


Figure 5.5: Correlation Matrix For Features Selected by Information Gain

5.4 Classification

The features-set proposed by this study is evaluated using different machine learning classifiers that include Decision Tree, K-Nearest Neighbor, SVM, and Naive Bayes for identification and categorization of Android malware. 10-fold cross validation technique is used for validating the result against each classifier. Table 5.2 shows the result of identification and categorization of Android malware against multiple machine learning classifiers.

Table 5.2: Evaluation of Proposed Features by Using Different Classifiers with 10-Fold Cross Validation

Classifier	**Output class**	**Precision**	**Recall**	**F1-Score**	**Accuracy**
Decision Tree	Adware	0.94	0.96	0.95	0.97
	Banking	1.00	1.00	1.00	
	Benign	0.99	0.96	0.97	
	Riskware	0.99	0.97	0.98	
	SMS	0.93	0.94	0.94	
	Avg Score	**0.97**	**0.97**	**0.97**	
K-Nearest Neighbor (KNN)	Adware	0.88	0.83	0.86	0.93
	Banking	0.99	0.98	0.99	
	Benign	0.95	0.94	0.95	
	Riskware	0.95	0.96	0.96	
	SMS	0.87	0.93	0.90	
	Avg Score	**0.93**	**0.93**	**0.93**	
SVM	Adware	0.92	0.91	0.92	0.97
	Banking	1.00	1.00	1.00	
	Benign	1.00	1.00	1.00	
	Riskware	0.99	0.97	0.98	
	SMS	0.92	0.96	0.94	
	Avg Score	**0.97**	**0.97**	**0.97**	
Naive Bayes	Adware	0.86	0.27	0.41	0.75
	Banking	0.82	0.99	0.90	
	Benign	0.89	0.78	0.83	
	Riskware	0.73	0.73	0.73	
	SMS	0.61	0.97	0.75	
	Avg	**0.78**	**0.75**	**0.72**	

5.5 Existing Feature-Set Evaluation

In [1] malware detection using features from `task_struct` is presented using 112 features. The work shows 94%-98% accuracy for binary classification (benign/malicious) of malware. The same set of features is used for malware categorization using 10-fold cross-validation techniques. The 112 features are reduced to 74 features after eliminating constant features, and then different machine learning classifiers are applied. Results are shown in Table 5.3.

Table 5.3: Existing Feature-set [1] Evaluation using 10-Fold Cross Validation Against Different Classifiers

Classifier	**Output class**	**Precision**	**Recall**	**F1-Score**	**Accuracy**
Decision Tree	Adware	0.67	0.71	0.69	0.73
	Banking	0.76	0.75	0.75	
	Benign	0.74	0.74	0.74	
	Riskware	0.71	0.74	0.73	
	SMS	0.80	0.73	0.77	
	Avg Score	**0.74**	**0.73**	**0.74**	
K-Nearest Neighbor (KNN)	Adware	0.70	0.68	0.69	0.73
	Banking	0.75	0.77	0.76	
	Benign	0.81	0.70	0.75	
	Riskware	0.70	0.71	0.71	
	SMS	0.71	0.80	0.75	
	Avg Score	**0.73**	**0.73**	**0.73**	
SVM	Adware	0.73	0.75	0.74	0.76
	Banking	0.75	0.73	0.74	
	Benign	0.86	0.79	0.82	
	Riskware	0.73	0.73	0.73	
	SMS	0.75	0.79	0.77	
	Avg Score	**0.76**	**0.76**	**0.76**	
Naive Bayes	Adware	0.67	0.19	0.29	0.39
	Banking	0.20	0.04	0.06	
	Benign	0.77	0.40	0.53	
	Riskware	0.73	0.34	0.47	
	SMS	0.28	0.99	0.43	
	Avg Score	**0.53**	**0.39**	**0.36**	

5.6 Comparison With Existing Study

This study proposed 98 features out of the 526 features for the detection and categorization of Android malware, which consist of seventy (70) newly identified features in addition to the features extracted by [1]. The experiments performed on the improved feature-set resulted in improved precision, recall, F1-score, and accuracy as compared to the existing feature-set for detection and categorization of Android malware. Table 5.4 shows the comparison of results achieved through existing feature-set and the improved feature-set proposed by this study. The result of existing study is taken from the evaluation of the feature presented in section 5.5. It can be observed from the Table 5.4 and Figure 5.6 that the improved feature-set proposed by this study greatly contributes to the detection and categorization of Android malware.

Table 5.4: Comparison Table of Feature-set Evaluation for Categorization of Android Malware

Feature set Proposed By	**Classifiers**	**Precision**	**Recall**	**F1-Score**	**Accuracy**
Wang et. al [1]	DT, KNN, SVM, NB	DT :0.74 KNN:0.73 SVM:0.76 NB :0.53	DT :0.73 KNN:0.73 SVM:0.76 NB :0.39	DT :0.74 KNN:0.73 SVM:0.76 NB :0.36	DT :0.73 KNN:0.73 SVM:0.76 NB :0.39
This study	DT, KNN, SVM, NB	DT : 0.97 KNN: 0.93 SVM: 0.97 NB : 0.78	DT : 0.97 KNN: 0.93 SVM: 0.97 NB : 0.75	DT : 0.97 KNN: 0.93 SVM: 0.97 NB : 0.72	DT : 0.97 KNN: 0.93 SVM: 0.97 NB : 0.75

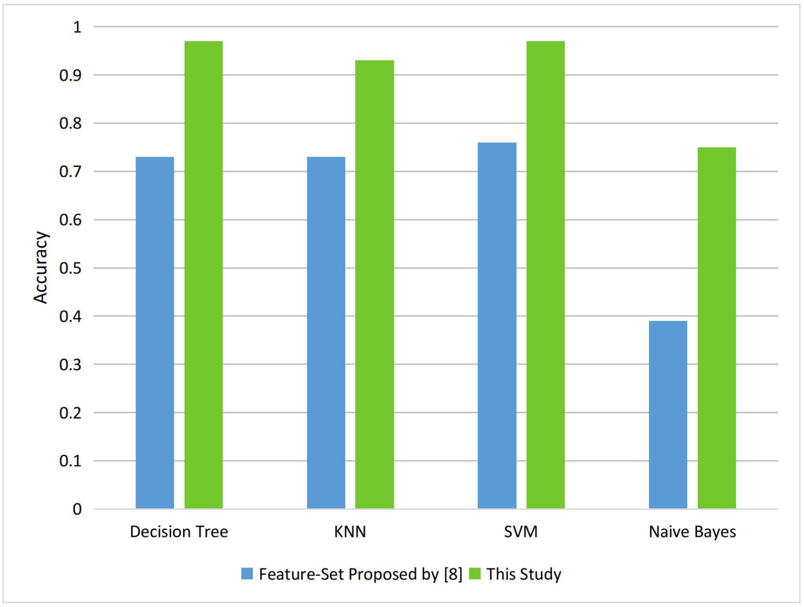


Figure 5.6: Performance Comparison of Android Malware Categorization Using Feature-Set Proposed by [1] and This Study

CHAPTER 6

CONCLUSION & FUTURE WORK

In this study, a feature extraction and classification framework is proposed for identification and categorization of Android malware. The proposed framework is implemented in Python, which is evaluated for the extraction of features against 2,500 Android applications selected from CICMalDroid 2020 dataset. The 2,500 Android applications belong to 5 categories (Adware, Banking Malware, Riskware, SMS Malware, and Benign Applications), and include 500 applications in each category. The feature extraction agent extracts features from the memory dump of Android device and stores the result into a csv file. The feature extraction process is performed four times against a single application from four different memory dumps captured at different times of application execution. The feature extraction agent extracted 526 features from the Android PCB (`task_struct`), which are then reduced to 98 features based on constant feature elimination, elimination of features based on Information Gain threshold, and features correlation. The 98 features consist of 70 newly identified features, while the remaining 28 features belong to the 112 features identified by [1]. By evaluating the proposed feature set with different machine learning classifiers, Decision Tree and SVM outperform the other classifiers with a precision = 0.97, Recall = 0.97, F1-Score = 0.97, and Accuracy = 0.97. When the same classifiers were applied to the existing feature set, the SVM classifier achieved the highest performance with a precision = 0.76, Recall = 0.76, F1-Score = 0.6, and Accuracy = 0.76. These results showed that the newly identified features have a high contribution to the detection and categorization of Android malware.

This study evaluated the effectiveness of improved features set extracted from Android PCB (`task_struct`) for the categorization of specific malware categories. In the future, we would like to extend our work for identification and categorization of obfuscated Android applications.

REFERENCES

[1] X. Wang, C. Li, Android malware detection through machine learning on kernel task structures, Neurocomputing 435 (2021) 126–150. doi:10.1016/j.neucom.2020.12.088.

[2] [Online], Number of smartphone subscriptions worldwide from 2016 to 2021, with forecasts from 2022 to 2027, Accessed: August. 28, 2022. URL `https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/`

[3] [Online], Mobile Operating System Market Share Worldwide, Accessed: August. 28, 2022. URL `https://gs.statcounter.com/os-market-share/mobile/worldwide/#monthly-202207-202207-bar`

[4] [Online], Average number of new Android app releases via Google Play per month from March 2019 to June 2022, Accessed: August. 27, 2022. URL `https://www.statista.com/statistics/1020956/android-app-releases-worldwide/`

[5] K. Shibija, R. V. Joseph, A machine learning approach to the detection and analysis of android malicious apps, in: 2018 International Conference on Computer Communication and Informatics (ICCCI), 2018, pp. 1–4. doi:10.1109/ICCCI.2018.8441472.

[6] [Online], Development of new Android malware worldwide from June 2016 to March 2020, Accessed: August. 28, 2022. URL `https://www.statista.com/statistics/680705/global-android-malware-volume/`

[7] P. Stirparo, I. N. Fovino, I. Kounelis, Data-in-use leakages from android memory test and analysis, in: 2013 IEEE 9th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob), 2013, pp. 701–708. doi:10.1109/WiMOB.2013.6673433.

[8] Z. Bazrafshan, H. Hashemi, S. M. H. Fard, A. Hamzeh, A survey on heuristic malware detection techniques, in: The 5th Conference on Information and Knowledge Technology, 2013, pp. 113–120. doi:10.1109/IKT.2013.6620049.

[9] R. Kumar, Z. Xiaosong, R. U. Khan, J. Kumar, I. Ahad, Effective and explainable detection of android malware based on machine learning algorithms, in: Proceedings of the 2018 International Conference on Computing and Artificial Intelligence, ICCAI 2018, Association for Computing Machinery, New York, NY, USA, 2018, p. 3540. doi:10.1145/3194452.3194465.

[10] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, L. Cavallaro, Droidsieve: Fast and accurate classification of obfuscated android malware, in: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, CODASPY '17, Association for Computing Machinery, New York, NY, USA, 2017, p. 309320. doi:10.1145/3029806.3029825.

[11] Y. Ding, X. Zhang, J. Hu, W. Xu, Android malware detection method based on bytecode image, Journal of Ambient Intelligence and Humanized Computing (2020) 1–10doi:10.1007/s12652-020-02196-4.

[12] L. Massarelli, L. Aniello, C. Ciccotelli, L. Querzoni, D. Ucci, R. Baldoni, Android malware family classification based on resource consumption over time, in: 2017 12th International Conference on Malicious and Unwanted Software (MALWARE), 2017, pp. 31–38. doi:10.1109/MALWARE.2017.8323954.

[13] H. Gao, S. Cheng, W. Zhang, Gdroid: Android malware detection and classification with graph convolutional network, Computers & Security 106 (2021) 102264. doi:10.1016/j.cose.2021.102264.

[14] M. Gohari, S. Hashemi, L. Abdi, Android malware detection and classification based on network traffic using deep learning, in: 2021 7th International Conference on Web Research (ICWR), 2021, pp. 71–77. doi:10.1109/ICWR51868.2021.9443025.

[15] A. S. Bozkir, E. Tahillioglu, M. Aydos, I. Kara, Catch them alive: A malware detection approach through memory forensics, manifold learning and computer vision, Computers & Security 103 (2021) 102166. doi:10.1016/j.cose.2020.102166.

[16] M. Hammad, J. Garcia, S. Malek, A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products, in: Proceedings of the 40th International Conference on Software Engineering, ICSE '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 421431. doi:10.1145/3180155.3180228.

[17] H. Cho, J. H. Yi, G.-J. Ahn, Dexmonitor: Dynamically analyzing and monitoring obfuscated android applications, IEEE Access 6 (2018) 71229–71240. doi:10.1109/ACCESS.2018.2881699.

[18] S. Mahdavifar, D. Alhadidi, A. A. Ghorbani, Effective and efficient hybrid android malware classification using pseudo-label stacked auto-encoder, Journal of Network and Systems Management 30 (1) (2022) 1–34. doi:10.1007/s10922-021-09634-4.

[19] X. Wang, C. Li, D. Song, Crowdnet: Identifying large-scale malicious attacks over android kernel structures, IEEE Access 8 (2020) 15823–15837. doi:10.1109/ACCESS.2020.2965954.

[20] L. Massarelli, L. Aniello, C. Ciccotelli, L. Querzoni, D. Ucci, R. Baldoni, Androdfa: android malware classification based on resource consumption, Information 11 (6) (2020) 326. doi:10.3390/info11060326.

[21] D. Carlin, P. OKane, S. Sezer, Dynamic analysis of malware using run-time opcodes, in: Data analytics and decision support for cybersecurity, Springer, 2017, pp. 99–125. doi:10.1007/978-3-319-59439-2_4.

[22] M. Benz, E. K. Kristensen, L. Luo, N. P. Borges, E. Bodden, A. Zeller, Heaps'n leaks: How heap snapshots improve android taint analysis, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 10611072. doi:10.1145/3377811.3380438.

[23] W. Zhang, H. Wang, H. He, P. Liu, Damba: Detecting android malware by orgb analysis, IEEE Transactions on Reliability 69 (1) (2020) 55–69. doi:10.1109/TR.2019.2924677.

[24] H. Alawneh, D. Umphress, A. Skjellum, Android malware detection using neural networks & process control block information, in: 2019 14th International Conference on Malicious and Unwanted Software (MALWARE), 2019, pp. 3–12.

[25] I. Krajci, D. Cummings, Android on x86: An Introduction to Optimizing for Intel Architecture, Apress, Berkeley, CA, 2013, Ch. History and Evolution of the Android OS, pp. 1–8. doi:10.1007/978-1-4302-6131-5_1.

[26] S. Khalid, F. B. Hussain, Evaluating dynamic analysis features for android malware categorization, in: 2022 International Wireless Communications and Mobile Computing (IWCMC), 2022, pp. 401–406. doi:10.1109/IWCMC55113.2022.9824225.

[27] [Online], Android - Statistics & Facts, Accessed: August. 28, 2022.
URL `https://www.statista.com/topics/876/android/`

[28] [Online], Android Architecture, Accessed: August. 29, 2022.
URL `https://source.android.com/docs/core/architecture`

[29] S. Sharma, R. Kumar, C. Rama Krishna, A survey on analysis and detection of android ransomware, Concurrency and Computation: Practice and Experience 33 (16) (2021) e6272. doi:10.1002/cpe.6272.

[30] [Online], Android Platform Architecture, Accessed: August. 29, 2022.
URL `https://developer.android.com/guide/platform`

[31] S. S. H. Shah, N. Jamil, A. u. R. Khan, Memory visualization-based malware detection technique, Sensors 22. doi:10.3390/s22197611.

[32] H.-H. Kim, M.-J. Choi, Linux kernel-based feature selection for android malware detection, in: The 16th Asia-Pacific Network Operations and Management Symposium, 2014, pp. 1–4. doi:10.1109/APNOMS.2014.6996540.

[33] F. Shahzad, M. Akbar, S. Khan, 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.

[34] J. Bellizzi, M. Vella, C. Colombo, J. Hernandez-Castro, Real-time triggering of android memory dumps for stealthy attack investigation, in: M. Asplund, S. Nadjm-Tehrani (Eds.), Secure IT Systems, Springer International Publishing, Cham, 2021, pp. 20–36.

[35] F. Shahzad, M. Shahzad, M. Farooq, In-execution dynamic malware analysis and detection by mining information in process control blocks of linux os, Information Sciences 231 (2013) 45–63, data Mining for Information Security. doi:10.1016/j.ins.2011.09.016.

[36] T. Panker, N. Nissim, Leveraging malicious behavior traces from volatile memory using machine learning methods for trusted unknown malware detection in linux cloud environments, Knowledge-Based Systems 226 (2021) 107095. doi:10.1016/j.knosys.2021.107095.

[37] N. Udayakumar, V. J. Saglani, A. V. Cupta, T. Subbulakshmi, Malware classification using machine learning algorithms, in: 2018 2nd International Conference on Trends in Electronics and Informatics (ICOEI), 2018, pp. 1–9. doi:10.1109/ICOEI.2018.8553780.

[38] [Online], Android Debug Bridge (adb), Accessed: August. 04, 2022. URL `https://developer.android.com/studio/command-line/adb`

[39] [Online], UI/Application Exerciser Monkey, Accessed: August. 04, 2022. URL `https://developer.android.com/studio/test/other-testing-tools/monkey`

[40] [Online], Android Asset Packaging Tool, Accessed: August. 04, 2022. URL `https://developer.android.com/studio/command-line/aapt2`

[41] [Online], LiME ˜ Linux Memory Extractor, Accessed: July. 17, 2022. URL `https://github.com/504ensicsLabs/LiME`

[42] [Online], Volatility Framework - Volatile memory extraction utility framework, Accessed: July. 17, 2022. URL `https://github.com/volatilityfoundation/volatility`

[43] A. Salah, E. Shalabi, W. Khedr, A Lightweight Android Malware Classifier Using Novel Feature Selection Methods, Symmetry 12 (5) (2020) 858. doi:10.3390/sym12050858.

[44] J. Abawajy, A. Darem, A. A. Alhashmi, Feature Subset Selection for Malware Detection in Smart IoT Platforms, Sensors 21 (4) (2021) 1374. doi:10.3390/s21041374.

[45] A. Feizollah, N. B. Anuar, R. Salleh, A. W. A. Wahab, A review on feature selection in mobile malware detection, Digital Investigation 13 (2015) 22–37. doi:10.1016/j.diin.2015.02.001.

[46] R. Jayanthi, L. Florence, Software defect prediction techniques using metrics based on neural network classifier, Cluster Computing 22 (1) (2019) 77–88.

APPENDIX A

LIST OF EXTRACTED ATTRIBUTES/FEATURES IN TASK_STRUCT CATEGORY-WISE

Table A.1: Task State Information

Task State attributes/features
1)task→exit_state 2)task→exit_code 3)task→exit_signal 4)task→pdeath_signal 5)task→jobctl 6)task→personality

Table A.2: Memory Information

Memory Information attributes/features
7)task → acct_rss_mem1 8)task → acct_timexpd 9)task → acct_vm_mem1 10)task → dirty_paused_when 11)task → maj_flt 12)task → min_flt 13)task → mm → arg_end 14)task → mm → arg_start 15)task → mm → brk 16)task → mm → context → ia32_compat 17)task → mm → context → lock → count → counter 18)task → mm → context → perf_rdpmc_allowed → counter 19)task → mm → def_flags 20)task → mm → end_code 21)task → mm → end_data 22)task → mm → env_end 23)task → mm → env_start 24)task → mm → exe_file → f_count → counter 25)task → mm → exe_file → f_cred → egid → val 26)task → mm → exe_file → f_cred → euid → val 27)task → mm → exe_file → f_cred → fsgid → val 28)task → mm → exe_file → f_cred → fsuid → val 29)task → mm → exe_file → f_cred → gid → val 30)task → mm → exe_file → f_cred → group_info → nblocks 31)task → mm → exe_file → f_cred → group_info → ngroups 32)task → mm → exe_file → f_cred → jit_keyring 33)task → mm → exe_file → f_cred → securebits 34)task → mm → exe_file → f_cred → sgid → val 35)task → mm → exe_file → f_cred → suid → val 36)task → mm → exe_file → f_cred → uid → val 37)task → mm → exe_file → f_cred → usage → counter

Continued on next page

Table A.2 – *Continued from previous page*

Memory Information attributes/features
38)task → mm → exe_file → f_cred → user → locked_shm 39)task → mm → exe_file → f_cred → user → mq_bytes 40)task → mm → exe_file → f_cred → user → unix_inflight 41)task → mm → exe_file → f_flags 42)task → mm → exe_file → f_inode → dirtied_time_when 43)task → mm → exe_file → f_inode → dirtied_when 44)task → mm → exe_file → f_inode → i_atime → tv_nsec 45)task → mm → exe_file → f_inode → i_atime → tv_sec 46)task → mm → exe_file → f_inode → i_blkbits 47)task → mm → exe_file → f_inode → i_blocks 48)task → mm → exe_file → f_inode → i_bytes 49)task → mm → exe_file → f_inode → i_count → counter 50)task → mm → exe_file → f_inode → i_ctime → tv_nsec 51)task → mm → exe_file → f_inode → i_ctime → tv_sec 52)task → mm → exe_file → f_inode → i_data → flags 53)task → mm → exe_file → f_inode → i_data → nrpages 54)task → mm → exe_file → f_inode → i_data → nrshadows 55)task → mm → exe_file → f_inode → i_data → writeback_index 56)task → mm → exe_file → f_inode → i_dio_count → counter 57)task → mm → exe_file → f_inode → i_flags 58)task → mm → exe_file → f_inode → i_fsnotify_mask 59)task → mm → exe_file → f_inode → i_generation 60)task → mm → exe_file → f_inode → i_gid → val 61)task → mm → exe_file → f_inode → i_ino 62)task → mm → exe_file → f_inode → i_mapping → flags 63)task → mm → exe_file → f_inode → i_mapping → nrpages 64)task → mm → exe_file → f_inode → i_mapping → nrshadows 65)task → mm → exe_file → f_inode → i_mapping → writeback_index 66)task → mm → exe_file → f_inode → i_mode 67)task → mm → exe_file → f_inode → i_mtime → tv_nsec 68)task → mm → exe_file → f_inode → i_mtime → tv_sec 69)task → mm → exe_file → f_inode → i_opflags 70)task → mm → exe_file → f_inode → i_rdev 71)task → mm → exe_file → f_inode → i_sb → cleancache_poolid 72)task → mm → exe_file → f_inode → i_sb → s_blocksize 73)task → mm → exe_file → f_inode → i_sb → s_blocksize_bits 74)task → mm → exe_file → f_inode → i_sb → s_count 75)task → mm → exe_file → f_inode → i_sb → s_dev 76)task → mm → exe_file → f_inode → i_sb → s_flags 77)task → mm → exe_file → f_inode → i_sb → s_iflags 78)task → mm → exe_file → f_inode → i_sb → s_magic 79)task → mm → exe_file → f_inode → i_sb → s_max_links 80)task → mm → exe_file → f_inode → i_sb → s_maxbytes 81)task → mm → exe_file → f_inode → i_sb → s_mode 82)task → mm → exe_file → f_inode → i_sb → s_quota_types 83)task → mm → exe_file → f_inode → i_sb → s_readonly_remount

Continued on next page

Table A.2 – *Continued from previous page*

Memory Information attributes/features
84)task → mm → exe_file → f_inode → i_sb → s_stack_depth 85)task → mm → exe_file → f_inode → i_sb → s_time_gran 86)task → mm → exe_file → f_inode → i_size 87)task → mm → exe_file → f_inode → i_state 88)task → mm → exe_file → f_inode → i_uid → val 89)task → mm → exe_file → f_inode → i_version 90)task → mm → exe_file → f_inode → i_writecount → counter 91)task → mm → exe_file → f_mapping → flags 92)task → mm → exe_file → f_mapping → host → dirtied_time_when 93)task → mm → exe_file → f_mapping → host → dirtied_when 94)task → mm → exe_file → f_mapping → host → i_blkbits 95)task → mm → exe_file → f_mapping → host → i_blocks 96)task → mm → exe_file → f_mapping → host → i_bytes 97)task → mm → exe_file → f_mapping → host → i_flags 98)task → mm → exe_file → f_mapping → host → i_fsnotify_mask 99)task → mm → exe_file → f_mapping → host → i_generation 100)task → mm → exe_file → f_mapping → host → i_ino 101)task → mm → exe_file → f_mapping → host → i_mode 102)task → mm → exe_file → f_mapping → host → i_opflags 103)task → mm → exe_file → f_mapping → host → i_rdev 104)task → mm → exe_file → f_mapping → host → i_size 105)task → mm → exe_file → f_mapping → host → i_state 106)task → mm → exe_file → f_mapping → host → i_version 107)task → mm → exe_file → f_mapping → i_mmap_rwsem → count 108)task → mm → exe_file → f_mapping → i_mmap_writable → counter 109)task → mm → exe_file → f_mapping → nrpages 110)task → mm → exe_file → f_mapping → nrshadows 111)task → mm → exe_file → f_mapping → page_tree → gfp_mask 112)task → mm → exe_file → f_mapping → page_tree → height 113)task → mm → exe_file → f_mapping → writeback_index 114)task → mm → exe_file → f_mode 115)task → mm → exe_file → f_owner → euid → val 116)task → mm → exe_file → f_owner → pid_type 117)task → mm → exe_file → f_owner → signum 118)task → mm → exe_file → f_owner → uid → val 119)task → mm → exe_file → f_path → dentry → d_flags 120)task → mm → exe_file → f_path → dentry → d_time 121)task → mm → exe_file → f_path → mnt → mnt_flags 122)task → mm → exe_file → f_pos 123)task → mm → exe_file → f_pos_lock → count → counter 124)task → mm → exe_file → f_ra → async_size 125)task → mm → exe_file → f_ra → mmap_miss 126)task → mm → exe_file → f_ra → prev_pos 127)task → mm → exe_file → f_ra → ra_pages 128)task → mm → exe_file → f_ra → start 129)task → mm → exe_file → f_version 130)task → mm → exec_vm 131)task → mm → flags 132)task → mm → highest_vm_end 133)task → mm → hiwater_rss 134)task → mm → hiwater_vm 135)task → mm → hugetlb_usage → counter

Continued on next page

Table A.2 – *Continued from previous page*

Memory Information attributes/features
136)task → mm → locked_vm 137)task → mm → map_count 138)task → mm → mm_count → counter 139)task → mm → mm_rb → rb_node → __rb_parent_color 140)task → mm → mm_rb → rb_node → rb_left → __rb_parent_color 141)task → mm → mm_rb → rb_node → rb_right → __rb_parent_color 142)task → mm → mm_users → counter 143)task → mm → mmap_base 144)task → mm → mmap_legacy_base 145)task → mm → mmap_sem → count 146)task → mm → mmap_sem → osq → tail → counter 147)task → mm → mmap → rb_subtree_gap 148)task → mm → mmap → shared → rb_subtree_last 149)task → mm → mmap → shared → rb → __rb_parent_color 150)task → mm → mmap → vm_end 151)task → mm → mmap → vm_file → f_count → counter 152)task → mm → mmap → vm_file → f_cred → jit_keyring 153)task → mm → mmap → vm_file → f_cred → securebits 154)task → mm → mmap → vm_file → f_flags 155)task → mm → mmap → vm_file → f_inode → dirtied_time_when 156)task → mm → mmap → vm_file → f_inode → dirtied_when 157)task → mm → mmap → vm_file → f_inode → i_blkbits 158)task → mm → mmap → vm_file → f_inode → i_blocks 159)task → mm → mmap → vm_file → f_inode → i_bytes 160)task → mm → mmap → vm_file → f_inode → i_flags 161)task → mm → mmap → vm_file → f_inode → i_fsnotify_mask 162)task → mm → mmap → vm_file → f_inode → i_generation 163)task → mm → mmap → vm_file → f_inode → i_ino 164)task → mm → mmap → vm_file → f_inode → i_mode 165)task → mm → mmap → vm_file → f_inode → i_opflags 166)task → mm → mmap → vm_file → f_inode → i_rdev 167)task → mm → mmap → vm_file → f_inode → i_size 168)task → mm → mmap → vm_file → f_inode → i_state 169)task → mm → mmap → vm_file → f_inode → i_version 170)task → mm → mmap → vm_file → f_mapping → flags 171)task → mm → mmap → vm_file → f_mapping → nrpages 172)task → mm → mmap → vm_file → f_mapping → nrshadows 173)task → mm → mmap → vm_file → f_mapping → writeback_index 174)task → mm → mmap → vm_file → f_mode 175)task → mm → mmap → vm_file → f_owner → pid_type 176)task → mm → mmap → vm_file → f_owner → signum 177)task → mm → mmap → vm_file → f_pos 178)task → mm → mmap → vm_file → f_ra → async_size 179)task → mm → mmap → vm_file → f_ra → mmap_miss 180)task → mm → mmap → vm_file → f_ra → prev_pos 181)task → mm → mmap → vm_file → f_ra → ra_pages 182)task → mm → mmap → vm_file → f_ra → start 183)task → mm → mmap → vm_file → f_version 184)task → mm → mmap → vm_page_prot → pgprot 185)task → mm → mmap → vm_pgoff

Continued on next page

Table A.2 – *Continued from previous page*

Memory Information attributes/features
186)task → mm → mmap → vm_rb → __rb_parent_color 187)task → mm → mmap → vm_start 188)task → mm → nr_pmds → counter 189)task → mm → nr_ptes → counter 190)task → mm → pgd → pgd 191)task → mm → pinned_vm 192)task → mm → shared_vm 193)task → mm → stack_vm 194)task → mm → start_brk 195)task → mm → start_code 196)task → mm → start_data 197)task → mm → start_stack 198)task → mm → task_size 199)task → mm → tlb_flush_batched 200)task → mm → tlb_flush_pending 201)task → mm → total_vm 202)task → mm → vmacache_seqnum 203)task → nivcsw 204)task → normal_prio 205)task → nr_dirtied 206)task → nr_dirtied_pause 207)task → nvcsw 208)task → pagefault_disabled 209)task → rss_stat → events 210)task → stime 211)task → stimescaled 212)task → tlb_ubc → flush_required 213)task → tlb_ubc → writable 214)task → usage → counter 215)task → utime 216)task → utimescaled 217)task → vmacache_seqnum

Table A.3: Signal Information

Signal Information attributes/features
218)task → sas_ss_size 219)task → sas_ss_sp 220)task → sighand → count → counter 221)task → signal → audit_tty 222)task → signal → audit_tty_log_passwd 223)task → signal → cgtime 224)task → signal → cinblock 225)task → signal → cmaj_flt 226)task → signal → cmaxrss 227)task → signal → cmin_flt 228)task → signal → cnivcsw 229)task → signal → cnvcsw 230)task → signal → coublock 231)task → signal → cputime_expires → stime 232)task → signal → cputime_expires → sum_exec_runtime 233)task → signal → cputime_expires → utime 234)task → signal → cputimer → checking_timer 235)task → signal → cputimer → cputime_atomic → stime → counter 236)task → signal → cputimer → cputime_atomic → sum_exec_runtime → counter 237)task → signal → cputimer → cputime_atomic → utime → counter 238)task → signal → cputimer → running 239)task → signal → cstime 240)task → signal → cutime 241)task → signal → flags 242)task → signal → group_exit_code 243)task → signal → group_stop_count 244)task → signal → gtime 245)task → signal → inblock 246)task → signal → ioac → cancelled_write_bytes

Continued on next page

Table A.3 – *Continued from previous page*

Signal Information attributes/features
247)task → signal → ioac → rchar 248)task → signal → ioac → read_bytes 249)task → signal → ioac → syscfs 250)task → signal → ioac → syscr 251)task → signal → ioac → syscw 252)task → signal → ioac → wchar 253)task → signal → ioac → write_bytes 254)task → signal → leader 255)task → signal → leader_pid → count → counter 256)task → signal → leader_pid → level 257)task → signal → live → counter 258)task → signal → maj_flt 259)task → signal → maxrss 260)task → signal → min_flt 261)task → signal → nivcsw 262)task → signal → notify_count 263)task → signal → nr_threads 264)task → signal → nvcsw 265)task → signal → oom_flags 266)task → signal → oom_score_adj 267)task → signal → oom_score_adj_min 268)task → signal → oublock 269)task → signal → pacct → ac_exitcode 270)task → signal → pacct → ac_flag 271)task → signal → pacct → ac_majflt 272)task → signal → pacct → ac_mem 273)task → signal → pacct → ac_minflt 274)task → signal → pacct → ac_stime 275)task → signal → pacct → ac_utime 276)task → signal → posix_timer_id 277)task → signal → prev_cputime → stime 278)task → signal → prev_cputime → utime 279)task → signal → real_timer → _softexpires → tv64 280)task → signal → real_timer → base → clockid 281)task → signal → real_timer → base → cpu_base → active_bases 282)task → signal → real_timer → base → cpu_base → clock_was_set_seq 283)task → signal → real_timer → base → cpu_base → cpu 284)task → signal → real_timer → base → cpu_base → max_hang_time 285)task → signal → real_timer → base → cpu_base → migration_enabled 286)task → signal → real_timer → base → cpu_base → nohz_active 287)task → signal → real_timer → base → cpu_base → nr_events 288)task → signal → real_timer → base → cpu_base → nr_hangs 289)task → signal → real_timer → base → cpu_base → nr_retries 290)task → signal → real_timer → base → index 291)task → signal → real_timer → base → offset → tv64 292)task → signal → real_timer → is_rel 293)task → signal → real_timer → node → expires → tv64 294)task → signal → real_timer → state 295)task → signal → sigcnt → counter 296)task → signal → stats_lock → seqcount → sequence 297)task → signal → stime 298)task → signal → sum_sched_runtime 299)task → signal → utime

Table A.4: Scheduling Information

Scheduling Information attributes/features
300)task → cputime_expires → stime 301)task → cputime_expires → sum_exec_runtime 302)task → cputime_expires → utime 303)task → dl → deadline 304)task → dl → dl_boosted 305)task → dl → dl_bw 306)task → dl → dl_deadline 307)task → dl → dl_density 308)task → dl → dl_new 309)task → dl → dl_period 310)task → dl → dl_runtime 311)task → dl → dl_throttled 312)task → dl → dl_timer → _softexpires → tv64 313)task → dl → dl_timer → base → clockid 314)task → dl → dl_timer → base → cpu_base → active_bases 315)task → dl → dl_timer → base → cpu_base → clock_was_set_seq 316)task → dl → dl_timer → base → cpu_base → cpu 317)task → dl → dl_timer → base → cpu_base → max_hang_time 318)task → dl → dl_timer → base → cpu_base → migration_enabled 319)task → dl → dl_timer → base → cpu_base → nohz_active 320)task → dl → dl_timer → base → cpu_base → nr_events 321)task → dl → dl_timer → base → cpu_base → nr_hangs 322)task → dl → dl_timer → base → cpu_base → nr_retries 323)task → dl → dl_timer → base → index 324)task → dl → dl_timer → base → offset → tv64 325)task → dl → dl_timer → is_rel 326)task → dl → dl_timer → node → expires → tv64 327)task → dl → dl_timer → node → node → __rb_parent_color 328)task → dl → dl_timer → state 329)task → dl → dl_yielded 330)task → dl → flags 331)task → dl → rb_node → __rb_parent_color 332)task → dl → runtime 333)task → flags 334)task → on_cpu 335)task → on_rq 336)task → policy 337)task → prev_cputime → lock → raw_lock → val → counter 338)task → prev_cputime → stime 339)task → prev_cputime → utime 340)task → prio 341)task → ptrace 342)task → rcu_read_lock_nesting 343)task → rt_priority 344)task → rt → schedtune_enqueued 345)task → rt → schedtune_timer → _softexpires → tv64 346)task → rt → schedtune_timer → base → clockid 347)task → rt → schedtune_timer → base → cpu_base → active_bases 348)task → rt → schedtune_timer → base → cpu_base → clock_was_set_seq 349)task → rt → schedtune_timer → base → cpu_base → cpu 350)task → rt → schedtune_timer → base → cpu_base → max_hang_time 351)task → rt → schedtune_timer → base → cpu_base → migration_enabled 352)task → rt → schedtune_timer → base → cpu_base → nohz_active 353)task → rt → schedtune_timer → base → cpu_base → nr_events 354)task → rt → schedtune_timer → base → cpu_base → nr_hangs 355)task → rt → schedtune_timer → base → cpu_base → nr_retries

Continued on next page

Table A.4 – *Continued from previous page*

Scheduling Information attributes/features
356)task → rt → schedtune_timer → base → index 357)task → rt → schedtune_timer → base → offset → tv64 358)task → rt → schedtune_timer → is_rel 359)task → rt → schedtune_timer → node → expires → tv64 360)task → rt → schedtune_timer → node → node → __rb_parent_color 361)task → rt → schedtune_timer → state 362)task → rt → time_slice 363)task → rt → timeout 364)task → rt → watchdog_stamp 365)task → sched_info → last_arrival 366)task → sched_info → last_queued 367)task → sched_info → pcount 368)task → sched_info → run_delay 369)task → se → avg → last_update_time 370)task → se → avg → load_avg 371)task → se → avg → load_sum 372)task → se → avg → period_contrib 373)task → se → avg → util_avg 374)task → se → avg → util_sum 375)task → se → depth 376)task → se → exec_start 377)task → se → load → inv_weight 378)task → se → load → weight 379)task → se → nr_migrations 380)task → se → on_rq 381)task → se → prev_sum_exec_runtime 382)task → se → sum_exec_runtime 383)task → se → vruntime 384)task → stack_canary 385)task → state 386)task → static_prio 387)task → wake_cpu 388)task → wakee_flip_decay_ts 389)task → wakee_flips

Table A.5: Process Credentials

Process Credentials attributes/features
390)task → cred → egid → val 391)task → cred → euid → val 392)task → cred → fsgid → val 393)task → cred → fsuid → val 394)task → cred → gid → val 395)task → cred → group_info → nblocks 396)task → cred → group_info → ngroups 397)task → cred → group_info → usage → counter 398)task → cred → jit_keyring 399)task → cred → securebits 400)task → cred → session_keyring → datalen 401)task → cred → session_keyring → flags 402)task → cred → session_keyring → gid → val 403)task → cred → session_keyring → last_used_at 404)task → cred → session_keyring → perm 405)task → cred → session_keyring → quotalen 406)task → cred → session_keyring → sem → count 407)task → cred → session_keyring → serial 408)task → cred → session_keyring → state 409)task → cred → session_keyring → uid → val 410)task → cred → session_keyring → usage → counter 411)task → cred → sgid → val 412)task → cred → suid → val 413)task → cred → uid → val 414)task → cred → usage → counter 415)task → cred → user → __count → counter

Continued on next page

Table A.5 – *Continued from previous page*

Process Credentials attributes/features
416)task → cred → user → epoll_watches → counter 417)task → cred → user → inotify_devs → counter 418)task → cred → user → inotify_watches → counter 419)task → cred → user → locked_shm 420)task → cred → user → locked_vm → counter 421)task → cred → user → mq_bytes 422)task → cred → user → pipe_bufs → counter 423)task → cred → user → processes → counter 424)task → cred → user → sigpending → counter 425)task → cred → user → uid → val 426)task → cred → user → unix_inflight 427)task → real_cred → egid → val 428)task → real_cred → euid → val 429)task → real_cred → fsgid → val 430)task → real_cred → fsuid → val 431)task → real_cred → gid → val 432)task → real_cred → group_info → nblocks 433)task → real_cred → group_info → ngroups 434)task → real_cred → group_info → usage → counter 435)task → real_cred → jit_keyring 436)task → real_cred → securebits 437)task → real_cred → session_keyring → datalen 438)task → real_cred → session_keyring → flags 439)task → real_cred → session_keyring → gid → val 440)task → real_cred → session_keyring → last_used_at 441)task → real_cred → session_keyring → perm 442)task → real_cred → session_keyring → quotalen 443)task → real_cred → session_keyring → sem → count 444)task → real_cred → session_keyring → serial 445)task → real_cred → session_keyring → state 446)task → real_cred → session_keyring → uid → val 447)task → real_cred → session_keyring → usage → counter 448)task → real_cred → sgid → val 449)task → real_cred → suid → val 450)task → real_cred → uid → val 451)task → real_cred → usage → counter 452)task → real_cred → user → __count → counter 453)task → real_cred → user → epoll_watches → counter 454)task → real_cred → user → inotify_devs → counter 455)task → real_cred → user → inotify_watches → counter 456)task → real_cred → user → locked_shm 457)task → real_cred → user → locked_vm → counter 458)task → real_cred → user → mq_bytes 459)task → real_cred → user → pipe_bufs → counter 460)task → real_cred → user → processes → counter 461)task → real_cred → user → sigpending → counter 462)task → real_cred → user → uid → val 463)task → real_cred → user → unix_inflight

Table A.6: I/O Statistics

I/O Statistics attributes/features
464)task → delays → blkio_count 465)task → delays → blkio_delay 466)task → delays → blkio_start 467)task → delays → flags 468)task → delays → freepages_count 469)task → delays → freepages_delay 470)task → delays → freepages_start 471)task → delays → swapin_count 472)task → delays → swapin_delay 473)task → ioac → cancelled_write_bytes 474)task → ioac → rchar 475)task → ioac → read_bytes 476)task → ioac → syscfs 477)task → ioac → syscr 478)task → ioac → syscw 479)task → ioac → wchar 480)task → ioac → write_bytes

Table A.7: Open File Descriptors

Open File Descriptors attributes/features
481)task → files → count → counter 482)task → files → fdt → close_on_exec 483)task → files → fdt → full_fds_bits 484)task → files → fdt → max_fds 485)task → files → fdt → open_fds 486)task → files → fdtab → close_on_exec 487)task → files → fdtab → full_fds_bits 488)task → files → fdtab → max_fds 489)task → files → fdtab → open_fds 490)task → files → next_fd 491)task → files → resize_in_progress

Table A.8: CPU Specific State

CPU Specific State attributes/features
492)task → thread → cr2 493)task → thread → debugreg6 494)task → thread → ds 495)task → thread → error_code 496)task → thread → es 497)task → thread → fpu → counter 498)task → thread → fpu → fpregs_active 499)task → thread → fpu → fpstate_active 500)task → thread → fpu → last_cpu 501)task → thread → fsbase 502)task → thread → fsindex 503)task → thread → gsbase 504)task → thread → gsindex 505)task → thread → io_bitmap_max 506)task → thread → iopl 507)task → thread → ptrace_dr7 508)task → thread → sp 509)task → thread → sp0 510)task → thread → trap_nr

Table A.9: Others

Others attributes/features
511)task → atomic_flags 512)task → btrace_seq 513)task → default_timer_slack_ns 514)task → gtime 515)task → last_switch_count 516)task → parent_exec_id 517)task → preempt_disable_ip 518)task → ptrace_message 519)task → pushable_tasks → prio 520)task → real_start_time 521)task → self_exec_id 522)task → sessionid 523)task → start_time 524)task → timer_slack_ns 525)task → trace 526)task → trace_recursion

APPENDIX B

LIST OF SELECTED FEATURES

Table B.1: Final feature set against each category

Task State features (01)
1)task → personality
Memory Info features (35)
2)task → usage → counter 3)task → acct_vm_mem1 4)task → vmacache_seqnum 5)task → utime 6)task → nivcsw 7)task → nr_dirtied 8)task → maj_flt 9)task → stimescaled 10)task → acct_rss_mem1 11)task → mm → mmap_legacy_base 12)task → mm → start_brk 13)task → mm → mmap → vm_file → f_inode → i_ino 14)task → mm → mmap → shared → rb → __rb_parent_color 15)task → mm → shared_vm 16)task → mm → exe_file → f_path → dentry → d_time 17)task → mm → mmap → vm_rb → __rb_parent_color 18)task → mm → mm_rb → rb_node → rb_right → __rb_parent_color 19)task → mm → exec_vm 20)task → mm → pgd → pgd 21)task → mm → mm_count → counter 22)task → mm → mmap → vm_page_prot → pgprot 23)task → min_flt 24)task → mm → mm_users → counter 25)task → mm → mmap → vm_end 26)task → mm → map_count 27)task → mm → hiwater_rss 28)task → mm → nr_ptes → counter 29)task → mm → mmap → shared → rb_subtree_last 30)task → mm → nr_pmds → counter 31)task → mm → vmacache_seqnum 32)task → mm → mmap → vm_file → f_ra → start 33)task → nvcsw 34)task → mm → mmap → vm_file → f_count → counter 35)task → mm → exe_file → f_inode → i_writecount → counter 36)task → mm → exe_file → f_count → counter
Signal Info features (24)
37)task → sas_ss_sp 38)task → signal → ioac → rchar 39)task → signal → ioac → wchar 40)task → signal → ioac → syscr 41)task → signal → sum_sched_runtime 42)task → signal → sigcnt → counter

Continued on next page

Table B.1 – *Continued from previous page*

43)task → signal → live → counter 44)task → signal → nr_threads 45)task → signal → ioac → syscw 46)task → signal → min_flt 47)task → signal → utime 48)task → signal → leader_pid → count → counter 49)task → signal → prev_cputime → stime 50)task → signal → prev_cputime → utime 51)task → signal → ioac → write_bytes 52)task → signal → oom_score_adj_min 53)task → signal → ioac → read_bytes 54)task → signal → ioac → syscfs 55)task → sighand → count → counter 56)task → signal → stats_lock → seqcount → sequence 57)task → signal → nvcsw 58)task → signal → nivcsw 59)task → signal → stime 60)task → signal → pacct → ac_majflt
Scheduling Info features (11)
61)task → dl → dl_timer → base → cpu_base → clock_was_set_seq 62)task → dl → dl_timer → base → cpu_base → nr_retries 63)task → dl → dl_timer → base → cpu_base → nr_events 64)task → prio 65)task → sched_info → run_delay 66)task → sched_info → pcount 67)task → se → vruntime 68)task → se → avg → util_sum 69)task → se → load → inv_weight 70)task → se → avg → load_sum 71)task → se → load → weight
Process Credentials features (07)
72)task → cred → session_keyring → last_used_at 73)task → cred → session_keyring → serial 74)task → cred → user → epoll_watches → counter 75)task → cred → usage → counter 76)task → cred → user → processes → counter 77)task → cred → user → mq_bytes 78)task → cred → user → pipe_bufs → counter
I/O Statistics features (11)
79)task → delays → blkio_start 80)task → delays → blkio_delay 81)task → delays → swapin_delay 82)task → delays → blkio_count 83)task → ioac → rchar 84)task → ioac → syscr 85)task → ioac → wchar 86)task → ioac → syscw 87)task → ioac → write_bytes 88)task → ioac → syscfs 89)task → ioac → read_bytes
Open file Descriptors features (09)
90)task → files → fdtab → close_on_exec 91)task → files → fdt → close_on_exec 92)task → files → fdt → open_fds 93)task → files → fdtab → open_fds 94)task → files → count → counter 95)task → files → fdt → full_fds_bits 96)task → files → next_fd 97)task → files → fdt → max_fds 98)task → files → fdtab → full_fds_bits

APPENDIX C

LIST OF COMMANDS

1) **To Check Whether Emulator/Device is Running/Connected**
```
adb devices
```

2) **To Boot the Emulator with Custom kernel**
```
path_to_android_sdk/emulator/emulator -avd <avd-name> -kernel
<kernel-image>
```

3) **To Copy File(s) to the Emulator**
```
adb push <file-to-copy> <target-location>
```

4) **To Install apk to the Emulator**
```
adb install <apk-file>
```

5) **Get Package Name of an APK using aapt(Android Asset Packaging Tool)**
```
aapt dump badging <apk-file> | awk -F" " '/package/ {print $2}'
| awk -F"'" '/name=/ {print $2}'
```

6) **To Run a Specific Application**
```
adb shell monkey -p <package-name> -c android.intent.categor
y.LAUNCHER 1
```

7) **To Find PID of a Specific Process by it's Package Name**
```
adb shell pidof <package-name>
```

8) **Configure and Load LiME to Send Memory dump to Host over TCP**
   - `adb forward tcp:4444 tcp:4444`
   - `adb shell insmod <path-to-LiME-Module> "path=tcp:4444 format=lime"`

9) **Receive Memory Dump on the Host**
```
nc localhost 4444 > <dump-file-name>
```

10) **To Issue Pseudorandom Events Against the Application**
```
adb shell monkey -p <package-name> -v <event-count>
```

11) **To Disable Device Admin for an Application**
```
adb shell pm disable-user <package-name>
```

12) **To Stop Execution of an Application**
```
adb shell am force-stop <package-name>
```

13) **To Uninstall an Application**
```
adb uninstall <package-name>
```

14) **To Power-off Emulator**
```
adb shell reboot -p
```

MS Thesis - Tracing Malicious Android Applications from Memory Dumps

by Khalid Imran

MS Thesis - Tracing Malicious Android Applications from Memory Dumps

ORIGINALITY REPORT

3%	%	3%	%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Saneeha Khalid, Faisal Bashir Hussain. "Evaluating Dynamic Analysis Features for Android Malware Categorization", 2022 International Wireless Communications and Mobile Computing (IWCMC), 2022 Publication	<1%
2	Pasquale Stirparo, Igor Nai Fovino, Ioannis Kounelis. "Data-in-use leakages from Android memory — Test and analysis", 2013 IEEE 9th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob), 2013 Publication	<1%
3	"Advances in Data and Information Sciences", Springer Science and Business Media LLC, 2022 Publication	<1%
4	Tomer Panker, Nir Nissim. "Leveraging malicious behavior traces from volatile memory using machine learning methods for	<1%

trusted unknown malware detection in Linux cloud environments", Knowledge-Based Systems, 2021
Publication

5	"Handbook of Big Data Analytics and Forensics", Springer Science and Business Media LLC, 2022 Publication	<1 %
6	"Proceedings of International Conference on Artificial Intelligence and Applications", Springer Science and Business Media LLC, 2021 Publication	<1 %
7	Barun Kumar Parichha. "Performance Analysis of Process Using Loadable Kernel Module (LKM)", 2015 Fifth International Conference on Communication Systems and Network Technologies, 2015 Publication	<1 %
8	Sanjay Madan, Sanjeev Sofat, Divya Bansal. "Tools and Techniques for Collection and Analysis of Internet-of-Things malware: A systematic state-of-art review", Journal of King Saud University - Computer and Information Sciences, 2022 Publication	<1 %
9	Deepa K., Radhamani G., Vinod P., Mohammad Shojafar, Neeraj Kumar, Mauro	<1 %

Conti. "Identification of Android malware using refined system calls", Concurrency and Computation: Practice and Experience, 2019
Publication

10	Haiyang Zhang, Qiaoyu Liang. "Red-Black Tree Used for Arranging Virtual Memory Area of Linux", 2010 International Conference on Management and Service Science, 2010 Publication	<1 %
11	"Cyber Threat Intelligence", Springer Science and Business Media LLC, 2018 Publication	<1 %
12	Arvind Mahindru, A. L. Sangal. "HybriDroid: an empirical analysis on effective malware detection model developed using ensemble methods", The Journal of Supercomputing, 2021 Publication	<1 %
13	Ashish Garg. "Using interpretable machine learning identify factors contributing to COVID-19 cases in the United States", Elsevier BV, 2022 Publication	<1 %
14	"Algorithms and Architectures for Parallel Processing", Springer Science and Business Media LLC, 2020 Publication	<1 %

15	"Security and Privacy in Communication Networks", Springer Science and Business Media LLC, 2018 Publication	<1 %
16	Ajay Kumara M.A., Jaidhar C.D.. "Automated multi-level malware detection system based on reconstructed semantic view of executables using machine learning techniques at VMM", Future Generation Computer Systems, 2018 Publication	<1 %
17	Gajendra Sharma. "Evaluation of Data Mining Categorization Algorithms on Aspirates Nucleus Features for Breast Cancer Prediction and Detection", International Journal of Education and Management Engineering, 2020 Publication	<1 %
18	"Deep Learning Applications for Cyber Security", Springer Science and Business Media LLC, 2019 Publication	<1 %
19	Yingwei Yu, Naizheng Bian. "An Intrusion Detection Method Using Few-Shot Learning", IEEE Access, 2020 Publication	<1 %

20	Wael F. Elsersy, Ali Feizollah, Nor Badrul Anuar. "The rise of obfuscated Android malware and impacts on detection methods", PeerJ Computer Science, 2022 Publication	<1 %
21	Farrukh Shahzad, M. Shahzad, Muddassar Farooq. "In-execution dynamic malware analysis and detection by mining information in process control blocks of Linux OS", Information Sciences, 2013 Publication	<1 %
22	Vikas Sihag, Manu Vardhan, Pradeep Singh. "A survey of android application and malware hardening", Computer Science Review, 2021 Publication	<1 %
23	Hyun-Il Kim, Moonyoung Kang, Seong-Je Cho, Sang-Il Choi. "Efficient Deep Learning Network with Multi-Streams for Android Malware Family Classification", IEEE Access, 2021 Publication	<1 %
24	Muhammad Ali, Stavros Shiaeles, Gueltoum Bendiab, Bogdan Ghita. "MALGRA: Machine Learning and N-Gram Malware Feature Extraction and Detection System", Electronics, 2020 Publication	<1 %

Exclude quotes	Off	Exclude matches	< 10 words
Exclude bibliography	On		