Utilizing Structural & In-execution PCB Information Analysis for Malware Detection on Linux based Smartphones & Computers

Submitted by
Farrukh Shahzad

As a partial requirement of for the degree of

Doctor of Philosophy
(2013)
Defence date:

International Reviewers:

Engin Kirda  Northeastern University in Boston, USA
Mohamed Younis  University of Maryland Baltimore, USA

Local Examiners:

Syed Ali Khayam  Region Manager at PLUMgrid
Fareed Zaffar  Lahore University of Management Sciences
Syed Affan Ahmed  FAST National University
Author’s Declaration

I, Farrukh Shahzad, hereby state that my PhD thesis titled “Utilizing Structural & In-execution PCB Information Analysis for malware detection on Linux based Smartphones & Computers” is my own work and it has not been previously submitted by me for taking partial or full credit for the award of any degree at this University or anywhere else in the world. If my statement is found to be incorrect, at any time even after my graduation, the University has the right to revoke my PhD degree.

-----------------------------------
Farrukh Shahzad

Dated:
Plagiarism Undertaking

I take full responsibility of the research work conducted during the PhD Thesis titled “Utilizing Structural & In-execution PCB Information Analysis for malware detection on Linux based Smartphones & Computers”. I solemnly declare that the research work presented in the thesis is done solely by me with no significant help from any other person; however, small help wherever taken is duly acknowledged. I have also written the complete thesis by myself. Moreover, I have not presented this thesis (or substantially similar research work) or any part of the thesis previously to any other degree awarding institution within Pakistan or abroad.

I understand that the management of Department of Electrical Engineering of National University of Computer and Emerging Sciences has a zero tolerance policy towards plagiarism. Therefore, I as an author of the above-mentioned thesis, solemnly declare that no portion of my thesis has been plagiarized and any material used in the Thesis from other sources is properly referenced. Moreover, the thesis does not contain any literal citing of more than 70 words (total) even by giving a reference unless I have the written permission of the publisher to do so. Furthermore, the work presented in the thesis is my own original work and I have positively cited the related work of the other researchers by clearly differentiating my work from their relevant work.

I further understand that if I am found guilty of any form of plagiarism in my thesis work even after my graduation, the University reserves the right to revoke my PhD degree. Moreover, the University will also have the right to publish my name on its website that keeps a record of the students who plagiarized in their thesis work.

--------------------------------
Farrukh Shahzad
Dated:

--------------------------------
Verified by Plagiarism Cell Officer
Dated:
I dedicate this thesis to Baji Fauzi, Ahsan Bhai and Khurram Bhai with abundance of love and gratitude.
Acknowledgements

I express my gratitude to Allah Almighty – the most gracious, the ever merciful – who granted me consistency and strength to complete this research endeavor. Afterwards, I would like to express my deep and special thanks to my PhD advisor Dr. Muddassar Farooq, for all his motivations, guidance, continuous support and precious suggestions. It was his inspiration that I started my PhD – he invited me to join his research center when I was working as a development team lead in Interactive Group after my MS(CE) – and due to his guidance and persistent interest I am able to complete the PhD research work. I hadn’t have a background of research before enrolment in PhD; consequently, he had to work hard to seed and polish my research capabilities. During my PhD, I also worked as project manager in his research center Next Generation Intelligent Research Center (nexGIN RC). So, I feel myself lucky that I got clear PhD research directions from him. In short, he is the person solely responsible for my PhD – from initiation to completion. I wish and pray him success in his future endeavors.

I am thankful to all my family members for their ever lasting support and prayers, in the pursuance of my academic milestones and my hard times of life. I salute the patience and sacrifice of my wife (Shazia Jilani) – for the time she spent alone in my hometown – during my PhD studies. Then, I would like to pay my heartiest thanks to Dr. Ayoob Alvi (Dean) and Dr. Aftab Maroof (Ex-Director Islamabad Campus) for playing an important role in confirmation of my PhD admission in FAST-NU. During my stay at FAST-NU, I am privileged to be introduced to Dr. Waseem Ikram (Director Islamabad Campus) and Dr. Anwar Majid Mirza (former HoD CS)). I appreciate their cooperation and encouragement throughout the PhD tenure.

Moreover, special thanks are due to all of my friends and colleagues of nexGIN RC especially M. Ali Akbar, M. Shahzad, Dr. Muhammad Saleem, Salman H. Khan and Sohail Masood Bhatti for their continuous support and help in bridging my gaps to achieve the milestones. I also thank to Ms Sada Yasmin and Syeda Saira Gillani - PhD scholars of M.A. Jinnah University, Islamabad - for proof reading of thesis.
Abstract

The advent of pervasive ubiquitous computing and advancement of wireless communication technologies has resulted in the proliferation of innovative mobile computing devices like tablets and smartphones. In consumer market and business community worldwide, smartphones have become the most reliable portable devices for Internet connectivity and sensitive data storage. As smartphones are becoming the core delivery platform for ubiquitous “connected customer services” paradigm; security threats and concomitant risks are also growing proportionally. Recent reports from security vendors highlight this emerging challenge. Additionally, smartphones pose limitations for security solution architects, such as limited computing power, memory, battery and peripherals etc. This makes the desktop security countermeasures infeasible for smartphone devices. Some known anti-malware commercial products for smartphones – by top ranked security vendors – are signature based and require continuous updating for latest malware detection. Moreover, these products are unable to detect the zero-day and polymorphic malwares for smartphones. Therefore, we argue that the domain of non-signature based anti-malware solutions for smartphones is open for research.

In this dissertation, a novel security framework is proposed for malware detection on Linux based computers and smartphones using different data mining approaches. At the outset, a technique is presented – by employing data mining methods – to detect malicious executables on Linux-based computers using static analysis of the structural information present in their headers. The empirical evaluation metrics (i.e. malware detection rate, false alarm rate, scalability of features, processing overheads and resilience to evasion) of the scheme bear out its feasibility for realtime deployment. Afterwards, we demonstrate an in-execution dynamic malware detection scheme on Linux using process control blocks of Linux kernel through time-series analysis of features and machine learning classification algorithms. The benchmark metrics indicate the suitability of scheme for malware detection at runtime. Moreover, we devise and demonstrate a comprehensive, lightweight hybrid framework comprising of static and dynamic (in-execution) malware detection techniques that are implemented at two different layers (formerly both components are presented individually to detect Linux malware). This hybrid framework is able to detect zero-day, polymorphic and packed malware on smartphones. It is demonstrated on OpenMoko smartphone – based on Angstorm distribution of embedded Linux OS. But, it is directly portable to superphones – launched by Canonical Ltd.
in 2013 – based upon Ubuntu Linux with support of native code execution. Finally, a malicious application detection framework is constituted for – world famous smartphone OS Android (with 64% market share) – by employing a novel scheme based on time-series analysis of process control blocks’ features, obtained from Android kernel. The classification results and benchmark indicators depict that the scheme is appropriate for detecting malicious applications on Android – during execution.
Contents

1 Introduction .................................................. 1
   1.1 Motivation ........................................... 2
   1.2 Problem Statement .................................. 3
   1.3 Research Methodology ................................ 4
   1.4 Major Contributions of the Dissertation .......... 5
      1.4.1 Analyzing Recent Smartphone Security Solutions & Formulating a Generic Security Framework .......... 5
      1.4.2 Novel Structural Feature-set Mining for Malicious Executables Detection ........................................ 5
      1.4.3 Kernel PCB Mining for In-execution Malware Detection ...... 6
      1.4.4 A Hybrid Security Framework for Linux based Smartphones . 6
      1.4.5 Malware Detection on Android using In-execution PCB Information Analysis ...................................... 7
   1.5 Organization of the Thesis ............................. 8
      1.5.1 Chapter 2: A Survey on Malicious Applications Analysis and Detection for Smartphones ................. 8
      1.5.2 Chapter 3: Using Structural Knowledge and Data Mining Methods To Detect New Malicious Executables ...... 9
      1.5.3 Chapter 4: In-Execution Dynamic Malware Analysis and Detection by Mining Information in PCB of Linux OS .... 9
      1.5.4 Chapter 5: A Hybrid Framework for Malware Detection on Smartphones using ELF Structural & PCB Runtime Traces . 10
      1.5.5 Chapter 6: TStructDroid: Realtime Malware Detection using Time-series Analysis of PCB on Android ............ 10
      1.5.6 Chapter 7: Conclusion and Future Work ................ 11

2 A Survey on Malicious Applications Analysis and Detection for Smartphones ................................. 12
   2.1 Introduction ........................................... 12
   2.2 Malicious Applications ................................ 14
      2.2.1 Types of Smartphone Malicious Applications .......... 15
      2.2.2 Infection Vectors & Methodologies ....................... 16
   2.3 Challenges for Malicious Applications Analysis & Detection .................................................. 17
      2.3.1 Generic Challenges for Malicious Applications Detection .................................................. 17
      2.3.2 Additional Challenges for Smartphone Platforms ........ 19
   2.4 Implementation Decisions ................................ 19
      2.4.1 Host-based vs Decoupled Security ....................... 19
      2.4.2 User space vs Kernel space ............................ 19
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4.3</td>
<td>Actual System vs Sandboxing/Emulation</td>
<td>20</td>
</tr>
<tr>
<td>2.4.4</td>
<td>Static Detection vs Dynamic Detection vs Hybrid Detection</td>
<td>20</td>
</tr>
<tr>
<td>2.4.5</td>
<td>One Class (Anomaly Detection) vs Two Class Classification</td>
<td>21</td>
</tr>
<tr>
<td>2.5</td>
<td>Generic Malicious Applications Detection Framework for Mobile Computing Devices</td>
<td>21</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Static Analysis and Detection Layer (SADL)</td>
<td>23</td>
</tr>
<tr>
<td>2.5.1.1</td>
<td>File Scanning Agent</td>
<td>23</td>
</tr>
<tr>
<td>2.5.1.2</td>
<td>Feature Processing Agent</td>
<td>23</td>
</tr>
<tr>
<td>2.5.1.3</td>
<td>Classification Agent</td>
<td>24</td>
</tr>
<tr>
<td>2.5.2</td>
<td>Dynamic Analysis and Detection Layer (DADL)</td>
<td>24</td>
</tr>
<tr>
<td>2.5.2.1</td>
<td>Process and System Monitor</td>
<td>24</td>
</tr>
<tr>
<td>2.5.2.2</td>
<td>Spatio-temporal Processing Agent (STPA)</td>
<td>24</td>
</tr>
<tr>
<td>2.5.2.3</td>
<td>Classification and Prediction Agent</td>
<td>24</td>
</tr>
<tr>
<td>2.6</td>
<td>Static Analysis Techniques for Detection of Security Threats and Privacy Leaks on Smartphones</td>
<td>25</td>
</tr>
<tr>
<td>2.6.1</td>
<td>Malicious Code Pattern Detection</td>
<td>25</td>
</tr>
<tr>
<td>2.6.2</td>
<td>Static Function Call Analysis</td>
<td>26</td>
</tr>
<tr>
<td>2.6.3</td>
<td>Static Permissions Leak Detection</td>
<td>26</td>
</tr>
<tr>
<td>2.7</td>
<td>Dynamic Analysis Techniques for Detection of Security Threats and Privacy Leaks on Smartphones</td>
<td>27</td>
</tr>
<tr>
<td>2.7.1</td>
<td>Information Flow Tracking</td>
<td>27</td>
</tr>
<tr>
<td>2.7.2</td>
<td>Dynamic Function Call Tracing</td>
<td>28</td>
</tr>
<tr>
<td>2.7.3</td>
<td>Runtime Permissions Leak Detection</td>
<td>28</td>
</tr>
<tr>
<td>2.7.4</td>
<td>Misbehavior analysis using power utilization patterns</td>
<td>29</td>
</tr>
<tr>
<td>2.7.5</td>
<td>System Performance/Behavior based Anomaly Detection</td>
<td>29</td>
</tr>
<tr>
<td>2.8</td>
<td>Security and Privacy Analysis &amp; Detection Tools for Smartphones</td>
<td>30</td>
</tr>
<tr>
<td>2.8.1</td>
<td>Woodpecker</td>
<td>30</td>
</tr>
<tr>
<td>2.8.2</td>
<td>Static Function Call Analysis for Collaborative Malware Detection on Android OS</td>
<td>31</td>
</tr>
<tr>
<td>2.8.3</td>
<td>Static Function Call Analysis using Centroid - Symbian</td>
<td>32</td>
</tr>
<tr>
<td>2.8.4</td>
<td>PiOS</td>
<td>33</td>
</tr>
<tr>
<td>2.8.5</td>
<td>SmartDroid</td>
<td>34</td>
</tr>
<tr>
<td>2.8.6</td>
<td>Multi-Level Anomaly Detector for Android Malware (MADAM)</td>
<td>34</td>
</tr>
<tr>
<td>2.8.7</td>
<td>Virus Meter - Battery Utilization patterns - Symbian</td>
<td>35</td>
</tr>
<tr>
<td>2.8.8</td>
<td>Energy-Greedy Anomalies &amp; Malware detection - Windows Mobile</td>
<td>36</td>
</tr>
<tr>
<td>2.8.9</td>
<td>Cloud-based Paranoid - Android</td>
<td>37</td>
</tr>
<tr>
<td>2.8.10</td>
<td>Crowdroid - System calls based decoupled security</td>
<td>38</td>
</tr>
<tr>
<td>2.8.11</td>
<td>Knowledge-based temporal abstraction - Android</td>
<td>38</td>
</tr>
<tr>
<td>2.8.12</td>
<td>Andromaly</td>
<td>39</td>
</tr>
<tr>
<td>2.8.13</td>
<td>Behavioral Misuse Detection - iPhone</td>
<td>40</td>
</tr>
<tr>
<td>2.8.14</td>
<td>TaintDroid</td>
<td>41</td>
</tr>
<tr>
<td>2.8.15</td>
<td>AppInspector</td>
<td>42</td>
</tr>
<tr>
<td>2.8.16</td>
<td>Android Application Sandbox (AASandbox)</td>
<td>42</td>
</tr>
<tr>
<td>2.8.17</td>
<td>XMAnDroid: Framework for Mitigation of Privilege Escalation Attacks</td>
<td>43</td>
</tr>
<tr>
<td>2.8.18</td>
<td>Quire</td>
<td>44</td>
</tr>
<tr>
<td>2.8.19</td>
<td>Stowaway - Android Permissions Demystified</td>
<td>45</td>
</tr>
</tbody>
</table>
2.8.20 Categorization of Android Applications using static features (CAASA) .................................................. 46
2.9 Misc. security & privacy solutions for smartphones ................................................. 46
2.9.1 Other Surveys on Smartphone Security ....................................................... 50
2.10 Conclusion ........................................................................................................ 51

3 ELF-Miner: Using Structural Knowledge and Data Mining Methods To Detect New Malicious Executables 53
3.1 Introduction ...................................................................................................... 53
3.2 Related Work ................................................................................................. 54
3.3 The Features’ set of ELF-Miner ................................................................. 55
3.3.1 Executable and linkable format (ELF) ....................................................... 55
3.3.2 ELF Structure-based Features’ set ............................................................. 56
3.4 ELF-Miner Framework ............................................................................... 58
3.5 ELF Malware Dataset .................................................................................. 58
3.6 Forensic Analysis of Benign and Malware ELF Executables ..................... 60
3.7 Classification Scheme ................................................................................. 64
3.7.1 Quantification ........................................................................................... 64
3.7.2 Preprocessing Features ............................................................................ 66
3.7.3 Classification ............................................................................................. 66
3.8 Experiments & Results ................................................................................. 67
3.9 Conclusions ................................................................................................. 71

4 In-Execution Dynamic Malware Analysis and Detection by Mining Information in Process Control Blocks of Linux OS 73
4.1 Introduction ...................................................................................................... 73
4.2 Forensic Analysis of Benign and Malicious Processes ......................... 76
4.2.1 Benign Processes ....................................................................................... 76
4.2.2 Malware Processes ..................................................................................... 76
4.2.3 Forensic Analysis ....................................................................................... 77
4.3 Dataset ............................................................................................................ 80
4.4 Architecture of Dynamic Malware Detection Framework ............... 80
4.4.1 Features Logger .......................................................................................... 81
4.4.2 Features Analyzer ...................................................................................... 81
4.4.3 Classification ............................................................................................. 85
4.5 Experiments and Results .............................................................................. 87
4.5.1 Overall classification accuracy using genetic footprint ................. 88
4.5.2 The impact of increasing detection time on the classification accuracy .......................................................................................... 88
4.5.3 Detection of backdoors-type processes ................................................. 90
4.5.4 False alarm rate of proposed system ..................................................... 91
4.5.5 Detection of tiny malicious processes .................................................... 91
4.5.6 Processing overheads of proposed scheme .......................................... 91
4.5.7 A comparison of the accuracy of the proposed scheme with other sequence calls based solutions .......................................................... 91
4.6 Evasion .......................................................................................................... 92
4.6.1 Robustness of proposed scheme to evasion attempts ....................... 93
4.6.2 Access restrictions to `task_struct` parameters .................................. 93
4.7 Related Work ................................................................................................ 94
4.8 Conclusion and Future Work ........................................ 95

5 A Hybrid Framework for Malware Detection on Smartphones using ELF Structural & PCB Runtime Traces ........................................... 97
  5.1 Introduction .......................................................... 97
  5.1.1 Organization of the chapter .................................... 99
  5.2 Related Work ........................................................ 99
  5.3 Polymorphic Malware Dataset ...................................... 101
    5.3.1 Malware Evolution: Silvio’s Case Study .................... 101
    5.3.2 Evolved Malware for ARM-V4t Platform ................... 102
    5.3.3 Malware Test & Training Dataset ........................... 103
      5.3.3.1 Singleton Test & Training-set .......................... 103
      5.3.3.2 Existere Test & Training-set .......................... 104
  5.4 A Hybrid Architecture of Malware Detection Framework .......... 104
  5.5 ELF Structural Tracer: Case Study ................................ 104
    5.5.1 EST prologue .................................................. 105
    5.5.2 ELF Structural Features Extractor ........................... 105
    5.5.3 Forensic Analysis ............................................. 105
    5.5.4 Pre-Processing Filters ....................................... 107
      5.5.4.1 Redundant Feature Removal (RFR) ....................... 107
      5.5.4.2 Dimensionality Reduction ............................... 107
  5.6 EST Classification & Results ..................................... 108
    5.6.1 Zero-day polymorphic malware detection .................... 109
    5.6.2 Existere polymorphic malware detection .................... 109
    5.6.3 Processing Overheads ....................................... 110
    5.6.4 EST features’ robustness analysis .......................... 110
  5.7 PCB Runtime Tracer: A Case Study ................................ 110
    5.7.1 PRT Features extractor ...................................... 111
      5.7.1.1 PCB runtime features - off-line analysis ............... 111
      5.7.1.2 Forensic investigation .................................. 111
      5.7.1.3 Features Selection ..................................... 113
  5.8 PRT Classification and Results .................................. 116
    5.8.1 Zero-day - polymorphic malware Detection .................. 116
    5.8.2 Multi-window processing & detection-accuracy ............. 116
    5.8.3 Existere malware - new variants detection .................. 117
    5.8.4 Packed Malware detection using PRT ........................ 117
    5.8.5 Processing overheads ....................................... 120
    5.8.6 Robustness to evasion attempts ............................ 120
  5.9 Conclusions and Future Work .................................... 120

6 TStructDroid: Realtime Malware Detection using Time-series Analysis of PCB on Android .................. 122
  6.1 Introduction ........................................................ 122
    6.1.1 Organization of the Chapter ................................ 124
  6.2 Android - Malware & Benign Datasets ............................ 124
    6.2.1 Benign Dataset .............................................. 124
    6.2.2 Malware Dataset ............................................. 125
    6.2.3 Creation of Training & Testing Datasets .................... 126
      6.2.3.1 Realtime Scenario ...................................... 126
List of Publications


List of Figures

2.1 Proposed generic malicious applications (security threats & privacy leaks) detection framework for smartphones ........................................ 22

3.1 Executable and linkable format (ELF) structural view [TI-Standards, 1993] ........................................................................... 56
3.2 Block diagram of ELF-Miner framework .................................................. 58
3.3 ELF header RA divergence graph for benign and malware files ...... 61
3.4 Sections’ frequency histogram for benign and malware files (%) ...... 61
3.5 Overall frequency of 7 sections in all malware (left) and benign (right) files .. ...................................................................... 62
3.6 Symbol Table RA divergence graph for benign and malware files ...... 62
3.7 Dynamic section RA divergence graph for benign and malware files . 63
3.8 Dynamic symbol section RA divergence for benign and malware files . 63
3.9 Relocation section RA divergence graph for benign and malware files 64
3.10 Segments frequency histogram for benign and malware files .. .. .. 65
3.11 Probability distribution plot of Information Gain for features extracted from multiple malware datasets’ .................................................. 66
3.12 The magnified ROC plots for scalability analysis of ELF-Miner features’ set ...................................................................... 69

4.1 Forensic analysis of benign and malicious processes ......................... 77
4.2 Histogram comparison of benign and malicious processes ................. 79
4.3 Block diagram of the dynamic malware detection framework ............ 81
4.4 Plots of various statistical aspects of short listed parameters ............... 82
4.5 Information Gain and Information Gain Ratio .................................. 86
4.6 Magnified ROC Plots for different classifiers .................................... 90

5.1 Block diagram of malware evolution process .................................. 101
5.2 A hybrid architecture for malware detection on smartphones .......... 104
5.3 Difference between benign and malware headers .......................... 106
5.4 DWT implementation using Daubechies-1 wavelet for feature selection 107
5.5 Difference between benign and malicious processes execution patterns 112
5.6 Plots of various statistical aspects of short listed parameters .......... 115
5.7 Accuracy plot for JRIP and J48 classifiers for multiple windows processing ................................................................. 117

6.1 Time-series Mean of Some Preliminary Features for Benign and Malicious Processes .............................................................. 129
<table>
<thead>
<tr>
<th>Figure Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2</td>
<td>Time-series Variance of Some Preliminary Features for Benign and Malicious Processes</td>
<td>129</td>
</tr>
<tr>
<td>6.3</td>
<td>Raw Time-series values of some shortlisted features</td>
<td>131</td>
</tr>
<tr>
<td>6.4</td>
<td>Time-series Cumulative Variance for Benign and Malicious Processes</td>
<td>133</td>
</tr>
<tr>
<td>6.5</td>
<td>Convergence of Cumulative Variance Rate for <code>task-&gt;usage.counter</code> (Theorem 1)</td>
<td>135</td>
</tr>
<tr>
<td>6.6</td>
<td>3D Visual Representation of Autoregressive Models for Benign and Malware processes</td>
<td>136</td>
</tr>
<tr>
<td>6.7</td>
<td>Block diagram of malicious applications detection on Android in real-time using process control blocks (task_struct) - process flow in user and kernel space</td>
<td>137</td>
</tr>
<tr>
<td>6.8</td>
<td>Information Theoretic Analysis of Frequency Components Related to Shortlisted Features Set $\mathcal{F}_{sel}$</td>
<td>138</td>
</tr>
<tr>
<td>6.9</td>
<td>Effect of change in Voting Window Size $W_{vote}$ (Real-time Scenario, $\delta t = 10$ms, $T = 10$ instances)</td>
<td>142</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Market Share of Smartphone Operating Systems (thousands of units) [Gupta et al., 2012] .......................................................... 13
2.2 Correlation Matrix of Tools and Features .................................................. 47
3.1 Features extracted from ELF files ............................................................... 56
3.2 Benign and malware file size normalization ................................................. 60
3.3 Dataset analysis for redundant and useless features .................................... 66
3.4 ELF malware detection Avg. Accuracy and ADD comparison of evolutionary and non-evolutionary algorithms ........................................ 68
3.5 Comparison of ELF-Miner with other static analysis based techniques (AUC) discussed in Related Work section ................................................ 68
3.6 Scalability analysis of ELF mined features’ set ........................................... 69
3.7 Malware detection capability of ELF features’ set with spoofed headers ............. 70
3.8 Comprehensibility analysis of rules generated by all evolutionary and non-evolutionary classifiers .................................................. 71
3.9 ELF sections & description [TI-Standards, 1993] [Haungs, 2009] .................. 72
4.1 Benign and malware file size distribution ..................................................... 80
4.2 Description of the fields constituting genetic footprint ................................... 85
4.3 Class noise of dataset used in experiments .................................................. 85
4.4 Accuracy results of J48 and J-Rip on each of the 10 folds ............................... 89
4.5 Comparison with different system call sequences based techniques .................. 92
4.6 Malware dataset with categories, names and file sizes [VX-Heavens, 2009] [Offensive-Computing, 2009] ........................................ 96
5.1 Overall classification results with Singleton training and test-set ................. 109
5.2 Overall classification results with Existere training and test-set ................. 109
5.3 Description of the fields constituting PCB runtime trace ............................ 113
5.4 Accuracy results of J48 and JRip on each of the 10 folds using Singleton training and test-set scheme ........................................ 118
5.5 Accuracy results of J48 and JRip on each of the 10 folds on Existere training and test-set scheme ........................................ 119
6.1 Short-listed task $\text{struct}$ parameters ($F_{\text{set}}$) of TstructDroid framework for classification purpose .................................................. 130
6.2 Classification Performance Results for Real-time & Cross-validation scenarios .......................................................... 140
6.3 System Performance Degradation Analysis on Android ................................ 143
List of Algorithms

5.1  Pseudo-code of evolved malware program template  . . . . . . . . . . 102
Nomenclature

ELF: Executable and Linkable Format (For Linux & UNIX-Like OS)
PCB: Process Control Blocks (Maintained inside the kernel of OS)
ARM: Advanced RISC Machine (Energy efficient processor architecture)
POC: Proof of concept
SADL: Static Analysis and Detection Layer
STPA: Spatio-temporal Processing Agent
DADL: Dynamic Analysis and Detection Layer
Task_Struct: The data structure used to maintain PCB in Linux OS
EST: ELF Structural Tracer
PRT: PCB Runtime Tracer
RFR: Redundant Feature Removal
DWT: Discrete Wavelet Transform
DCT: Discrete Cosine Transform
Chapter 1

Introduction

In recent era, mobile handheld devices, including smartphones and tablets, typically have the computing power which is comparable to that of the desktop computers of the last decade of the previous century. As a result, smartphones have integrated an essential enabler for accessing “connected services” in a ubiquitous manner. Smartphones are now enabling customers to access m-government services, to stay connected with the family and friends on the social media, to do electronic transactions and e-commerce, to participate in audio/video live streaming conferences, to remotely attend e-learning classes and to diagnose and monitor patients in an e-health (or m-health) environment. As a consequence, the number of smartphones users have exponentially grown globally. In a recent survey of Gartner Research, conducted in 2011 (4th quarter), smartphone market grew by 47% as compared to year 2010 (4th quarter) [Egham, 2012]. In the beginning of 2012, mobile phones market had reached 419.1 million units, out of which 144.3 million are smartphones [Gartner, 2012]. Mobile phone market was anticipated to reach 645 million till the end of 2012 which would surpass the number of personal computers worldwide [Gartner, 2012]. In a report by Kaspersky Lab [Product-News, 2012], 16% of smartphone users store and transfer private documents from their smartphones, 53% use smartphones to send or receive emails, and 47% stay connected on the social networks (Facebook, Twitter etc.) using their phones. Typically, 62% of smartphone users browse the Internet using their smartphones. Google’s Android based smartphones dominate the market with a maximum of 64.1% overall share, while Apple’s iOS based smartphones have 18.8% share in the second quarter of 2012 [Gupta et al., 2012]. In case of Android only – Linux kernel based widespread mobile OS – more than 3000% increase in malicious software was observed in 2011 and a sixfold increase in 2012 [Trend-Micro, 2012]. The top security officials at Symantec and Mcafee confirm [NDTV-Gadgets, 2012] that the increased market share of smartphones has made them an ideal target for hackers and malicious software writers. They are now shifting their focus to smartphones instead of desktop computers.

All known malware\(^1\) categories of desktop systems can be seen on smartphone platform. But the most common, well-known, and widespread malware categories on smartphone are spyware, bots [Zhou and Jiang, 2012], adware [MilaParkour, 2012], trojans [Krebs, 2012], worms [Adeel and Tokarchuk, 2011] and rootkits [Bickford

\(^1\)The terms malware, malicious applications and malicious software are used interchangeably in this dissertation.
1. Introduction

Common challenges for security researchers and experts on desktop systems are: (1) polymorphic and zero-day malware detection, (2) higher malware detection accuracy, (3) lower false alarm rate (preferably zero), (4) performance degradation of system (due to the installation and execution of anti-malware solutions), (5) detection delay, (6) robustness of the featureset (used to discriminate benign and malware) against evasion attempts of crafty malware writer. In addition to these challenges and requirements, smartphone platforms introduce some additional challenges and bottlenecks for security vendors and researchers which are: (1) battery constraints (malware detection systems should be energy efficient), (2) limited processing power, (3) limited memory, and (4) processors architecture issues (smartphones usually use a processor with lower energy requirements i.e. ARM). In view of this, smartphone security has become a colossal challenge for security researchers worldwide. Top vendors of security products have launched anti-malware solutions for smartphones but most of these products are signature based and require continuous updating; moreover, they are unable to detect zero-day and polymorphic malware.

To solve this dilemma, researchers have proposed several non-signature based malware detection frameworks; presented on actual devices, sandboxes and emulators [Blasing et al., 2010] to operate on both user and kernel space of smartphones’ OS. Broadly speaking, existing security solutions for smartphones can be classified into two categories based on the type of analysis done at both kernel and user space: (1) static analysis, and (2) dynamic and behavioral analysis.

Security frameworks based upon static analysis, can detect malware before their execution on the device [Grace et al., 2012b]. They usually disassemble executables and installable packages to analyze code patterns [Blasing et al., 2010], static function calls [Egele et al., 2011] or over-privilege of applications [Felt et al., 2011a] to detect malicious applications. Alternatively, malware detection techniques based upon dynamic or behavioral analysis typically utilize features obtained from processes at runtime. They use techniques like information flow tracking [Enck et al., 2010], runtime instruction traces [Portokalidis et al., 2010], dynamic function call tracing [Burguera et al., 2011], runtime permissions leaks [Bugiel et al., 2011], power utilization patterns [Liu et al., 2009], system performance or behavioral features [Shabtai et al., 2010b] [Damopoulos et al., 2012] to identify malicious or misbehaving programs. Some researchers have proposed hybrid solutions based upon combinations of static and dynamic malware detection components. Classical machine learning techniques [Damopoulos et al., 2012][Schmidt et al., 2009a], clustering algorithms [Schmidt et al., 2009b] and control flow graphs [Egele et al., 2011] are frequently employed for anomaly detection and two-class classification.

Though most of the proposed solutions are host based, some cloud based solutions (decoupled security framework) [Portokalidis et al., 2010] have also been proposed to protect against malware. Typically, these approaches extract featureset from smartphones, compress them and transfer to cloud for further processing, classification, signature and non-signature based malware analysis and detection.

1.1 Motivation

The facts and figures – from leading organizations in computers and mobile security market – indicate the escalating rise in malware on smartphones. Most of these
malicious software are developed: (1) to steal private information of mobile users, and (2) to take control of smartphone devices (illegitimate administrative permissions) and use them to launch or execute unauthorized activities or malicious codes in a secret and stealthy way. Moreover, legacy style anti-malware (signature based) solutions are insufficient to detect zero-day malicious software (on the day of their launch) and polymorphic variants of existing malicious applications on smartphones. These facts and figures are the motivation to design a lightweight, non-signature based security framework on smartphones that can efficiently detect zero-day and polymorphic malware using unconventional features of malicious executables and programs.

1.2 Problem Statement

Contrary to traditional desktop computer systems, mobile devices have well-known limitations: limited computational power, processors with low power footprints, limited memory and low-power battery etc. [Oberheide and Jahanian, 2010] those should be given a serious consideration while designing malware detection systems for them. In spite of the underlying constraints, we believe that an efficient security and protection mechanism for smartphones can be designed using known data mining, knowledge engineering, and machine learning techniques and algorithms. In the context of smartphones' security, core objectives of this dissertation are:

1. To design a lightweight, reliable security solution that is capable of detecting zero-day malicious applications before execution as a first-line-of-defense; like traditional anti-malware solutions. This solution should use the features extracted from application’s executable binary.

2. In case, the malicious application evades before-execution malware detection component; an in-execution dynamic malware detection solution should detect it during the course of its execution as a second-line-of-defense. The dynamic solution should use the programs’ credentials to discriminate the malicious processes from benign ones.

The newly designed security framework for resource constrained smartphone platforms should be able to meet the following basic requirements.

1. The security framework should be able to detect zero-day malware (previous unknown).

2. It should be able to detect polymorphic and repacked variants of existing malware.

3. Malware detection rate should be higher (near 100%); malware shouldn’t be able to escape from the security system.

4. A malware detection system should ideally have a zero false alarm rate. It should not wrongfully prompt end users to kill or quarantine the legitimate applications.

5. The detection delay of malware detection system should be suitable for its realtime deployment on resource constrained mobile devices.
6. The processing overheads of the newly designed system should be relatively small.

7. It should not adversely effect the performance of exiting applications – installed on the system – to a level where they become useless.

8. Ideally, the framework and features’ set (used for non-signature based malware detection) should be resilient to evasion attempts made by the hackers and crafty malware writers.

9. The proposed framework should be tested, using real world malware samples, on a real smartphone platform instead of some emulator or sandbox.

1.3 Research Methodology

This section sequentially provides, the milestones of research methodology, followed by us as a roadmap for the design of the security framework for mobile computing devices.

1. In the initial phase, the security requirements, challenges, strategies and constraints on different smartphone platforms – are explored. A theoretical, generic security framework for mobile computing devices is proposed, that consists of two components: static and dynamic in-execution analysis. This requirement analysis and critical review (of legacy solutions) phase is a prerequisite for presenting a realtime security framework for smartphones.

2. In view of the initial phase, a prototypical manifestation of security component is designed and implemented, which is based on the static structural analysis of executables for Linux platform. The existing Linux malware are used for the analysis purpose.

3. On the other hand, another archetype demonstration is proposed, designed and implemented based upon dynamic in-execution analysis of process control blocks of Linux kernel. Linux kernel is chosen because of its open source nature.

4. Afterwards, both components are integrated to present a comprehensive security architecture for Linux based smartphone OpenMoko – with a customized set of malware (cross compiled and evolved) for ARM architecture. The framework is tested and validated within constrained environment of smartphone.

5. Finally, the dynamic in-execution malware detection framework is extended, redesigned and validated for Android OS (based upon process control block of Android kernel). Android is most popular smartphone OS with 65% market share. Real-world Linux malware are used to determine the feasibility of proposed solutions for malware detection on smartphones.
1.4 Major Contributions of the Dissertation

Design, formulation and empirical evaluation of a malware detection framework is a challenging and exigent task on traditional desktop systems; but it becomes more complex, convoluted and multifaceted on constrained platform of mobile computing devices. In this dissertation, a hybrid approach is proposed that utilizes structural and runtime PCB information analysis for malware detection on smartphones. Little work using structural knowledge for malicious executables detection (only on windows platform) is available in literature. Moreover, no one in research community has used them on Linux or smartphone platform. It is pertinent to mention that the principal component of our approach – using the runtime analysis of time-series process control blocks – has never been utilized on any platform (desktop or mobile) for detecting malware. In this context, a significant effort is done in multiple dimensions and substantial contributions are made in the area of malware detection. This section provides a brief and concise description of key contributions of four years research on malware detection techniques.

1.4.1 Analyzing Recent Smartphone Security Solutions & Formulating a Generic Security Framework

Mobile handheld devices like tablets and smartphones have gained immense popularity in recent years, due to an increase in their computational capabilities and ubiquitous services. In the requirement analysis and critical review phase of the dissertation, recent trends and advancements in worldwide smartphone market are explored in multiple dimensions such as usage trends of smartphone users, market share of different mobile operating systems, multiple infection vectors and emerging malware categories, mandatory requirements and challenges for designing anti-malware solutions for smartphones. In view of this, critical review and analysis of state-of-the-art malware detection and privacy preserving techniques is performed. To provide guidelines for future security solutions on mobile devices, a generic security solution (based upon a combination of static and dynamic analysis techniques) is formulated.

1.4.2 Novel Structural Feature-set Mining for Malicious Executables Detection

It is demonstrated that static analysis of structural information mined from executable files (without executing them) can be used to detect zero-day malicious executables. In this context, a solution is proposed to detect malicious executables using static analysis of structural information, extracted from different headers of the Executable and Linkable (ELF) files, and test it using real Linux malware (taken from known online malware repositories). This malware detection framework a prototypical manifestation on Linux – directly portable to other UNIX-like OS without modifications and cross compilation – as first-line-of-defense. It is also a candidate solution for malware detection using structural features on smartphones as well. To elaborate the discriminatory nature of structural features of malicious and benign files, a comprehensive forensic study is performed. Moreover, to substantiate the classification potential, information theoretic measures are used to quantify the
1. Introduction

featureset. Evolutionary and classical machine learning classifiers are employed for extensive evaluation. This study clearly exhibits that the detection accuracy, false alarm rate, scalability, resilience to evasion, tiny processing overheads and detection delay make this solution suitable for realtime deployment.

1.4.3 Kernel PCB Mining for In-execution Malware Detection

Malware detection techniques based on static analysis are prone to polymorphism and code obfuscation techniques; therefore some malware may escape from first-line-of-defense i.e. static analysis based tools and are allowed to execute. To catch the malicious programs in-execution, an effective and efficient archetype demonstration solution is presented on Linux, based upon their runtime behavior using genetic footprint that consists of novel featureset of process control blocks (task_struct) in the kernel of OS. This solution is compatible with other UNIX-like OS e.g. Solaris etc. To the best of our knowledge, nobody – in security community before us – used PCB features for detection of malicious processes. A series of systematic investigations is formed on PCB featureset forensic, time-series, and statistical analysis to establish the differentiation of benign and malicious processes, short-listing features for classification, and to corroborate the features and classifier selection etc. The classification results using machine learning algorithms demonstrate the high detection accuracy and low false alarm rate. Moreover, the built-in resilience of PCB features for crafty malware writers who went to evade them at runtime, and small processing overheads of the scheme are the major contributions which distinguish it from other dynamic malware detection techniques available in literature. The scheme acts as second-line-of-defense in security and protection systems.

1.4.4 A Hybrid Security Framework for Linux based Smartphones

The proof-of-concept of individual components, based on static and dynamic analysis are previously presented on desktop Linux OS. Therefore, we believe that a comprehensive – hybrid of both static and in-execution dynamic analysis – malware detection framework should be evaluated on smartphone platform. With this motivation, in this dissertation, a hybrid security framework is developed that utilizes structural and in-execution PCB information analysis for malware detection on Linux based smartphones. The key features of this hybrid framework are given below:

1. The static analysis based component extracts structural traces when the executable is launched for execution. RFR filters are used for preprocessing, and discrete wavelet transform (DWT) is applied for dimensionality reduction. Only approximation components of DWT are provided to machine learning algorithms for classification purpose.

2. In case, the malicious executables evade initial analysis and are launched by OS kernel as a process; then their PCB runtime traces are extracted (significant features are short-listed in a separate case study using time-series analysis) during execution. The results of time-series classification predictions – by
machine learning algorithms – are reported on the basis of a majority vote in
a specific time window.

3. A malware dataset is prepared for ARM Linux architecture using customized
malware evolution and cross compilation of existing Linux x86 malware.

4. To demonstrate the scenario of zero-day malware detection – for both the
static and in-execution dynamic analysis components – evolved samples of an
individual malware family are separated to form a single testset fold (ten folds
in total).

5. Alternatively, 10% samples of each malware family are combined in a fold (ten
folds in total) to present the scenario of polymorphic malware detection.

6. The framework is developed with the intention of making it generic and open
for future research. It is presented on ARM Linux based smartphone but it is
portable to all Linux based smartphones that support native code execution.

7. The framework is directly portable to next generation mobile computing de-
vices i.e. Ubuntu Linux based Superphone presented by Canonical Ltd. in
2013 [Superphone, 2013] with the capabilities of a laptop computer that sup-
ports all native core applications without Java virtual machine. Moreover,
Samsung plans to launch its famous Galaxy Nexus smartphone with Ubuntu
Linux OS in the first/second quarter of 2013.

8. The component of framework based upon runtime PCB analysis is portable
to TIZEN [Tizen, 2013] as well – an open source OS for smartphone – that is
developed by the joint venture of Samsung and Intel Corporation with multiple
devices support e.g. smartphones, tablets, smart televisions and in-vehicle
infotainment devices etc.

### 1.4.5 Malware Detection on Android using In-execution
PCB Information Analysis

Android is most widespread and popular smartphone operating system. Recent sur-
vvey of security product vendors report that malware threads on Android platforms
have escalated sharply; because an increasing number of smartphone users are us-
ing their devices for storing privacy-sensitive information and performing financial
transactions etc. Therefore, the solution is presented on Android platform as well.
The key contributions are as follows:

1. A realtime malware detection framework for Android platform is presented
that performs dynamic analysis of smartphone applications and detects ma-
licious activities through in-execution monitoring of process control blocks in
Android kernel.

2. Using information theoretic analysis, time-series feature logging, segmentation
and frequency component analysis of data, and a machine learning classifier,
this framework is able to detect real world malware applications on Android.
3. To demonstrate the realtime scenario of zero-day malware detection on Android, malicious applications have been tested using a real world dataset. In this scenario, a machine learning classifier is trained using the PCB traces of all benign and malware processes except of the process that needs to be tested. In addition to this, a standard ten-fold cross validation methodology is also employed to test the framework in a classical validation environment.

4. Deployment feasibility and performance of the solution is measured using standard metrics: malware detection and false alarm rate, processing overhead, detection delay and overall performance degradation of smartphones. The results indicate that the solution is feasible for realtime deployment on Android for detecting malicious applications.

This framework should be deployed in conjunctions with some Kernel Rootkits protection system to safeguard the process control blocks in Android kernel and smooth execution of the framework for malicious applications’ detection.

1.5 Organization of the Thesis

The rest of the dissertation is organized into five chapters. A brief description of each chapter is provide here.

1.5.1 Chapter 2: A Survey on Malicious Applications Analysis and Detection for Smartphones

The rationale of the survey is to explore and identify the requirements, challenges and principles for designing security framework on smartphones. The merits and demerits of existing techniques are also studied and evaluated in detail – in this requirement analysis and critical review phase.

To achieve this objective, initially, various types of malicious applications and infection vectors on smartphones, are enumerated. In order to develop detection techniques, it is relevant to understand the important challenges; therefore, they are enumerated as challenges for malicious applications analysis, detection and implementation decisions. Afterwards, a conceptual and generic detection framework for mobile malware is presented. The framework augments the system wide understanding of malware analysis and detection techniques. Recently published techniques, based on static and dynamic malware analysis and detection on smartphones are presented, analyzed, and categorized. Moreover, recently developed malware detection frameworks and tools that utilize these techniques are described and reviewed as well. Furthermore, a brief overview of other surveys and related solutions on security and privacy is also provided. Finally, the chapter is concluded with an outlook towards future directions of detecting mobile malware. This chapter is based on the paper:

1.5.2 Chapter 3: Using Structural Knowledge and Data Mining Methods To Detect New Malicious Executables

This chapter presents a prototypical manifestation for detecting malicious executable on Linux based smartphones using structural information. This proof-of-concept solution is presented for Linux and all Unix-Like operating systems. To this end, first we present and evaluate other malware detection methods based on static analysis. Afterwards, the structural features' extracted from Linux executable files (ELF) are presented. After the description of features, the dataset collected from known malware repositories vx heavens and offensive computing is briefly discussed. To highlight the discrimination between benign and malicious executables’ structural features, comprehensive forensic evaluation is performed. Moreover, the classification potential of features is quantified using information theoretic measures and then pre-processing filters are employed to eliminate the redundant and useless features. Furthermore, an extensive evaluation using classical rule based machine learning and evolutionary classifiers is done – to select the best classifier for our system. Finally, experiments are performed to do the scalability analysis of structural headers and their resilience against evasion attempts of the crafty malware writer. This chapter is based on the paper:


1.5.3 Chapter 4: In-Execution Dynamic Malware Analysis and Detection by Mining Information in PCB of Linux OS

This chapter presents archetype demonstration of detecting malicious programs on Linux using process control of OS kernel. To this end, the information in the task structure is used by doing a time series forensic study of selected malware and benign processes. In the next step, the characteristics of benign and malware dataset are discussed that are used for experiments. Afterwards, the functionality of three modules of the proposed scheme is explained: (1) features logging, (2) features selection, and (3) classification. The design of the scheme is based on systematic investigation and evaluations. Later on, the results of experiments are described and in doing so intriguing insights are provided about the behavior of the proposed scheme. Finally, the performance comparison of the proposed scheme is provided with multiple system calls sequence based solutions and a detailed description of the strengths of the proposed scheme against evasion attempts is presented. Additionally, a brief overview of the previous work related to behavior-based malware detection is provided and finally the chapter concludes with an outlook to future work. This chapter is based on papers:

- Farrukh Shahzad, Muhammad Shahzad, Muddassar Farooq, In-Execution Dynamic Malware Analysis and Detection by Mining Information in Process
1.5.4 Chapter 5: A Hybrid Framework for Malware Detection on Smartphones using ELF Structural & PCB Runtime Traces

In this chapter, a comprehensive, hybrid framework for malware detection on smartphones is presented – which is based on static analysis of ELF structural traces and in-execution dynamic analysis of PCB runtime traces. To this end, our hybrid framework consists of two components: (1) ELF Structural Tracer (EST) and (2) PCB Runtime Tracer (PRT). EST extracts structural traces of executables to devise a non-signature-based zero-day and polymorphic malware-detection scheme for Linux based smartphones. PCB runtime tracer, on the other hand, detects unknown malicious processes and their polymorphic variants at runtime. To demonstrate a proof-of-concept (POC), existing desktop Linux malware are cross compiled for ARM based Linux on smartphones. Two different types of datasets – singleton and existence – are prepared to demonstrate the zero-day and polymorphic malware detection scenarios. Afterwards, details of the integrated framework for smartphone platform are presented; both the schemes are presented as separate case studies. The case study of malware detection using EST consists of brief forensic analysis, classification scheme and result analysis, and scalability analysis etc. Similarly, the case study of PRT consists of segments – from feature extraction to time-series analysis for feature selection and result analysis in different experimental paradigms. Processing overheads, robustness to evasion and detection delays of both schemes in architecture are also discussed. Finally, the chapter is concluded with a review of architecture along with future directions. This chapter is based on the paper:


1.5.5 Chapter 6: TStructDroid: Realtime Malware Detection using Time-series Analysis of PCB on Android

This chapter provides a realtime malicious applications detection framework on Android – world’s most popular operating system for smartphones – using time-series analysis of process control blocks during the execution of applications. In the beginning, an itemized description of the related work in the field of dynamic malware analysis on smartphone platforms is provided – to identify shortcomings of recently proposed frameworks. Afterwards, methodology to collect the traces of our benign and malicious applications is described. To demonstrate the scenarios of: (1)
malicious applications detection on Android in realtime, and (2) standard research methodology of classification in different folds of dataset. The major components of our proposed TStructDroid framework are presented and the working of each component is discussed in detail. The description of the framework components is accompanied by discussion, on theoretical and empirical validation of the reasoning behind each milestone. Finally, the performance evaluation results are presented, in particular the processing overhead, detection delay and degradation in response time of applications executing in parallel on smartphone. Finally, the chapter is concluded with an outlook towards future directions. This chapter is based on the paper:


1.5.6 Chapter 7: Conclusion and Future Work

This chapter delineates the essence of key findings and achievements of this dissertation. It is concluded that static analysis of structural features and dynamic, time-series analysis of runtime PCB features is suitable for malware detection on Linux based mobile computing devices. The resultant security frameworks are more efficient than their counterparts in terms of malware detection accuracy, false alarm rate, processing overhead, detection delay, performance degradation of host platform, resistance against evasion attempts of crafty malware writers etc. Finally, the areas and research direction are highlighted that can be taken in near future in the avenues of smartphone security research.
Chapter 2

A Survey on Malicious Applications Analysis and Detection for Smartphones

2.1 Introduction

Smartphones are becoming the core delivery platform of ubiquitous “connected customer services” paradigm; as a consequence, they are attractive targets of malicious intruders (or imposters). Researchers have realized that classical signature-based anti-malware techniques are not capable of providing efficient and effective detection tools against novel, zero-day and polymorphic malware for resource constrained smartphones; therefore, in last couple of years unconventional (non-signature) intelligent solutions, based on behavioral analysis (static or dynamic) have been proposed. In this survey chapter, we provide an overview of the recently proposed static and dynamic malicious application detection techniques for smartphones.

In recent years, mobile hand held devices like smartphones and tablets have been popularized with comparable computational power – as that of the x86 computer systems of the last decade of the twentieth century. As an upshot, the smartphones vendors have integrated mechanism and applications for accessing connected services in smartphones in a ubiquitous manner. This rapid increase in popularity have also raised sever security threats for these mobile computing devices.

The major security vendors have recently reported an alarming rise in the malware attacks on smartphones. These malicious applications\(^1\) are a source of serious concern for two reasons: (1) they have the ability to take control of the phone and perform unauthorized activities in a stealthy (unnoticed) manner; and (2) they can access a user’s private information and leak/sell it to his adversary or different advertising agencies. In the “mobile threat report 2011” by Juniper Networks Inc. [Juniper-Report, 2012], the overall growth for all mobile operating systems, in their malware sample database, during 2011 was 61%. In case of Android, the malware samples increased from 400 to 13302 (an increase of 3325%) during the last seven months of 2011. The distribution of malware samples for different operating systems in 2010 was: JavaMe:70.3%, Windows Mobile:1.4%, Symbian:27.4%, BlackBerry:0.4%, and Android:0.4%. In comparison, the unique malware samples

\(^1\)We will use the terms malware and malicious applications interchangeably in this chapter.
collected in 2011 show a totally different distribution: JavaMe 41%, Windows Mobile 0.7%, Symbian 11.5%, BlackBerry 0.2% and Android 46.7%. To conclude, the number of threats on Android OS have significantly increased. The report also provides a taxonomy of the malware observed in 2011: 63.39% are Spyware, 36.43% are SMS Trojan, 0.09% are SMS-Flooder, and 0.09% are Worms. On Android platform, fake installers are the biggest infection vector (56% of the total threats). The Internet threat report of Symantec Corporation [Paul Wood et al., 2012] also confirms a dramatic increase in smartphone malware. Their analysts have categorized the malware on the basis of functionality and potential risks: 28% malware are data collection applications, 24% send smartphone stolen contents to the remote hosts, 25% are used for location tracking, 7% change the device settings and 16% are traditional threats (virus and worms etc.). F-Secure Labs in their mobile threats report 2012 [Mobile-Threat-Report, 2012] report that only 10 new malware families (along with their variants) were known by the beginning of 2011. Only within a year, 37 new malware families and their variants have been discovered. They report that malware writers are exploring new infection vectors and are focusing on evading malware detection techniques. Take the example of already known malware families – DroidKungFu, GinMaster, and Fakeinst umbrella – that are using cryptography and randomization techniques for evasion. Malware writers have also used images to hide the malicious code in them e.g. FakeRegSMS. Furthermore, they report that profit motivated threats (34%) are significantly higher as compared to non-profit threats (15%). A sixfold increase in malicious applications (175,000 in September compared with 30,000 in June) has been reported on Android platform in the third quarter of 2012 [Trend-Micro, 2012].

The exponential increase in malware on smartphones has started an anti-malware product race within security vendors worldwide. Most of them have enhanced their signature based anti-virus products [Best-AntiVirus, 2012] for smartphones. As a result, the phones are protected against malware whose signatures exist in the database. Malware forensic experts analyze malware and generate signatures that are inserted into the threats databases. The antivirus applications on an end user’s phone use the (malware) signature databases to detect malware. The process of creating a malware signature on the basis of the domain knowledge of a forensic expert is time consuming, challenging and error-prone. The true challenge for them is: to create a generic signature to achieve 0% false alarm rate (it should be able to detect mutated variants of the same malware and moreover it should not detect legitimate programs as malware). Malware writers typically release their malware in families consisting of multiple variants of a single malware having minor changes in

Table 2.1: Market Share of Smartphone Operating Systems (thousands of units) [Gupta et al., 2012]

<table>
<thead>
<tr>
<th>OS</th>
<th>2nd Quarter 12</th>
<th>2nd Quarter 11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Units</td>
<td>Market Share (%)</td>
</tr>
<tr>
<td>Android</td>
<td>98,529.30</td>
<td>64.1</td>
</tr>
<tr>
<td>iOS</td>
<td>28,935.00</td>
<td>18.8</td>
</tr>
<tr>
<td>Symbian</td>
<td>9,071.20</td>
<td>5.9</td>
</tr>
<tr>
<td>RIM</td>
<td>7,991.20</td>
<td>5.2</td>
</tr>
<tr>
<td>Bada</td>
<td>4,208.80</td>
<td>2.7</td>
</tr>
<tr>
<td>Microsoft</td>
<td>4,087.00</td>
<td>2.7</td>
</tr>
<tr>
<td>Others</td>
<td>863.3</td>
<td>0.6</td>
</tr>
<tr>
<td>Total</td>
<td>153,686.10</td>
<td>100</td>
</tr>
</tbody>
</table>
functionality, structure and/or code. Every week, the security vendors mostly receive a large number of malware variants out of them only few dozens are new malware families [McAfee-Threat-Report, 2012]; therefore, the capability to automate the process of malware analysis and detection is gaining momentum. Broadly speaking, malware analysis and detection techniques for smartphones can be categorized into two classes: (1) static detection techniques, and (2) dynamic detection techniques. Static analysis is performed on the disassembled binary files without executing the malware. On the other hand, dynamic analysis is performed by monitoring the malware and/or its effects on a system during or after execution of the malware.

The major contributions of this survey chapter are:

1. Focusing on niche and state-of-the-art malicious applications detection techniques for smartphone platforms.\(^1\)

2. Formulating a generic detection framework for malicious applications that will be used as an architecture guideline for mapping (or understanding) existing anti-malware security products.

3. Discussing the emerging and new (recently proposed) detection techniques that are used by smartphone security and privacy analysts.

4. Surveying (at length) recently proposed tools – using the above-mentioned techniques – with an aim to understand their relative merits and shortcomings.

The rest of the chapter is organized as follows. In Section 2.2, we enumerate various types of malicious applications and the infection vectors they exploit on smartphones. In order to develop detection techniques, it is relevant to understand the important challenges; therefore, we enumerate them in Sections 2.3 and 2.4. In Section 2.5, we present a generic detection framework for mobile malware. The framework augments the system wide understanding of malware analysis and detection techniques. We also present, analyze and categorize the latest published techniques for static and dynamic malware detection on smartphones in Section 2.6 and Section 2.7 respectively. In Section 2.8, we describe and review the recently developed malware detection frameworks and tools that utilize these techniques. Other surveys and misc. security and privacy solutions on smartphones are presented in Section 2.9. Finally, we conclude the chapter with an outlook towards the future directions of detecting mobile malware.

### 2.2 Malicious Applications

In this section, we describe the types of malicious applications that have plagued the smartphone systems over the last few years. We also enumerate the commonly used methodologies and typical infection vectors utilized for spreading malware infections to new (uninfected) smartphones.

\(^1\)The most of existing survey lack comprehensive treatment on mobile malware and instead focus broader smartphone security.
2.2 Malicious Applications

2.2.1 Types of Smartphone Malicious Applications

In order to make the chapter self-contained, we provide a brief overview of different types of malicious applications in this section. An interested reader can refer to [Zhou and Jiang, 2012] [Miller et al., 2012] [Damopoulos et al., 2011] [Schmidt and Albayrak, 2008] [Bose et al., 2008] [Szor, 2005] for an in-depth discussion on smartphone malicious applications.

A malicious code that usually spreads by exploiting vulnerabilities in the network services is termed as a worm. The worms usually run as independent executables and make copies of themselves on the networked machines. The oldest known worm is the famous Morris worm [Szor, 2005]. On smartphones, the first known worm Caribe (Worm.SymbOS.Cabir) was released in June 2004 [Gostev, 2006]. Other notable worms include iOS Ikee Worm [Porras et al., 2009] and Commwarror [Adeel and Tokarchuk, 2011].

Viruses infect normal processes on a system and use them to execute their malicious code. They usually spread through sharing of infected programs. Duts [F-Secure, 2012] is one of the well known viruses for smartphones.

Trojan Horses pose themselves as legitimate and productive applications and they execute their malicious codes in the background without user’s knowledge. They often spread by infecting other programs on the new system, and are often used as a gateway for installing additional malware on the infected system. Most malware for smartphones are trojans. Some well known trojan horses include Trojan-SMS.AndroidOS.FakePlayer [Symantec, 2012a], ZeuS Trojan [Krebs, 2012], Trojan-SMS.AndroidOS.Foncy [Maslennikov, 2011] and Skull.D [F-Secure, 2007].

Spyware present themselves as useful programs but their primary objective is to steal sensitive information about a user such as passwords, emails, user surfing behavior, credit card information etc. Some notable spyware include GPSSMSSpy and Nickyspy [Zhou and Jiang, 2012]. It is important to realize that most of mobile malware writers focus on stealing a user’s private information and use it to characterize her/his behavior in order to spam her/him with relevant advertisements. In 2011 alone, 63.39% of mobile malware were Spyware [Juniper-Report, 2012]. As a consequence, protecting a user’s privacy and detecting malicious applications have become synonymous in the security jargon.

Sometimes the primary purpose of an attacker is to get a large number of computing resources at her/his disposal. This is usually achieved by a specific type of malware known as Bot. Bots (infected computers) dial home to a master bot that then controls all the bots and issues commands to them. The whole network of bots and master bot is known as botnet. Typical uses of botnets include denial of service attacks, spamming, fraudulent activities etc. Anserverbot, Nickybot and Beanbot [Zhou and Jiang, 2012] are representative malware of this category.

A rootkit hides a malicious application on a system by modifying a system’s kernel such that the system API calls are instrumented (the logging and other similar operation binaries are replaced) [Bickford et al., 2010]. TDL, ZeroAccess [Symantec, 2012b], ITFUNZ and Z4Mod [MilaParkour, 2012] are examples of rootkits infecting mobile devices.

It is important for a hacker to ensure access to a hacked system even if the present vulnerability is patched in the future. Backdoors serve exactly this purpose. The presence of a backdoor allows unauthorized access to a system that enables a remote attacker to run commands on the system, usually by opening a port and waiting
Modern malicious application writers have established an underground malware mafia industry to quickly accumulate wealth. *URL injectors* replace actual search results and web links in a user’s browser with alternate links, web pages and sales pages that transfer revenue to the malware author or her/his associates. *Adware* programs usually display ads and pop ups trying to force a user to click on them and buy the products being advertised using affiliate accounts. Affiliate advertising brings in money for the malware writer. *SslCrypt on Symbian OS* [Fortinet, 2011] and *Toplank (Counterclank)* [Mcafee, 2012], *Plankton* [MilaParkour, 2012] are examples of adware. Some malware applications (*Money Stealers*) perform money sending actions such as sending premium SMS messages to a malware author or her/his associates account without user’s consent. *Foncy on Android* [Maslennikov, 2011] is one such example. A collection of latest malware applications for Android, iPhone and Symbian platforms is available on the website [www.contagiodump.blogspot.com](http://www.contagiodump.blogspot.com).

### 2.2.2 Infection Vectors & Methodologies

To protect malware infections, it is important to understand how malicious applications infect a smartphone and spread it to other systems. We now introduce the most important infection vectors for smartphones only.

The major source of automated infection spreading are exploitable vulnerabilities in applications that listen to network traffic or process incoming traffic. Malicious applications writers can automate the process of exploitation for such scenarios; as a result, infection spreads very quickly in a short time (specially) when the vulnerable services are common across many systems on the Internet. This infection methodology is utilized by worms.

When a user is browsing the Internet, malicious websites can exploit vulnerabilities in her/his browser to download and install malicious code on her/his machine without her/his consent [Juniper-Report, 2012]. A user doesn’t necessarily need to visit malicious websites for such attacks. The techniques – cross site scripting (XSS), clickjacking, script injection and iframe injection – are typically used by attackers to spread malicious code to the visitors of innocuous websites. This methodology is usually coupled with social engineering exploits to trick a user to go to the malicious website and become infected.

Social Engineering techniques use (or misuse) a user’s trust paradigm or naivety to entice him/her into actions that make it possible for a malicious application to do the malicious activity. For example, a link in an email from a coworker’s email address could actually lead to a malicious website that installs malicious code on a user’s system. In early days of Internet, emails and chat messengers were the primary technologies used for such attacks. With the evolution of social networks, expanded online social circles, interaction with strangers and user provided content publication, novel social engineering attacks are becoming straightforward. A tweet on a famous hashtag with a malicious link and a promise of never-seen-before video of a celebrity is all it takes nowadays to entice thousands (if not millions) of susceptible users to download and install a video plug-in that is actually a trojan.

Vulnerabilities in the operating system allow an un-privileged program to gain
enough privileges to infect the entire system, including the operating system files, and the directories of other users; as a result, infection spreads to all users of the machine. Removable Media such as SSD cards may contain infected files which may execute when the removable media is connected to the system. The malware on the infected system tries to copy itself to the new removable media devices and thus spread further. On a network with shared resources such as shared file servers, email clients etc., the malware can spread by copying itself to writable locations (or infecting executable files existing there), and then infecting the systems of the users who access those resources.

Smartphone platforms usually allow access to various protected system resources using a permission model. Over-privileged smartphone applications can become an important infection vector that can allow an application to leak private information or execute tasks not authorized by the user. It is also possible that another malicious application can use a legitimate privileged application as a deputy for an attack if the legitimate application provides an open interface for invocation. This can lead to privilege escalation attacks as well.

An attacker can setup rogue public WiFi access points, luring users to enjoy free Internet access [Juniper-Report, 2012]. By pointing to a rogue DNS server controlled by the attacker, the users can be redirected to malicious websites which can infect the system through drive-by download methodology described earlier. Bluetooth and MMS functionalities are ubiquitous. Bluetooth [Haataja et al., 2011] connections and MMS have the capability to help a malware leverage the system vulnerabilities to install itself on the system. Commwarrior [Adeel and Tokarchuk, 2011] is an example of malware that spreads through MMS messages.

Some mobile platforms (e.g. Android) support open application distribution architecture. A user can download applications from virtually any place on the Internet without the need of getting them signed from a competent authority. The systems with open architecture are highly susceptible to rogue application distribution sites that trick users to download rogue applications and install them [Juniper-Report, 2012].

2.3 Challenges for Malicious Applications Analysis & Detection

After discussing the common types of malicious applications and relevant infection vectors that help in propagation of the malware, we now list a number of challenges that are faced by security experts working on designing detection frameworks for malicious applications.

2.3.1 Generic Challenges for Malicious Applications Detection

We first present the challenges for malicious applications detection that are common to both desktop and mobile operating systems.

**Packers.** Malware try to evade static detection using packers [Zhou and Jiang, 2012][Moser et al., 2007]. Packing involves either compression or encryption (or both) of binary executables. This compressed and encrypted image is loaded at
runtime to perform malicious operations. Static malware detection schemes – like n-gram analysis – usually fail to correlate the packed malicious binary as malware because their content patterns are significantly altered during the packing process.

Polymorphic malware. Polymorphic malware [Moser et al., 2007] are similar to packed malware because it contains an encrypted malicious payload and a decryption routine. However, unlike packers, polymorphic malware try to evade detection by re-encrypting its contents, using a different key, every time it executes.

Metamorphic malware. Metamorphic malware modify their code by rewriting themselves on each infection [Moser et al., 2007] that makes their detection a challenge. The common techniques used to transform the code are: (1) register renaming (using different CPU registers in instructions); (2) code permutation (intelligently ordering instructions to preserve the semantic outcome/result); (3) code expansion (using more instructions to do the same thing); and (4) code compression and garbage code insertion (such as adding NOPs). Dynamic detection of such malware, using advanced virtualization techniques, yield better detection results.

System Performance Degradation. Processing overheads are more important during dynamic detection because they directly contribute towards deteriorating completion time of processes and this may have severe consequences especially for time-critical realtime processes.

Detection delay. A detection scheme should have low detection delay for a better user experience. Similarly, the memory cost of maintaining detection data structures should be within acceptable limits on resource constrained smartphones. Early detection of an executing malware is a challenge because delay might lead to an infected system that might end up in an unrecoverable state. To mitigate this, security solutions keep a log of runtime changes done on the memory and disk, and used rollback mechanisms to undo these changes (disinfection) after detection.

High False Alarm Rate. A malware detection system should ideally have a zero false alarm rate; otherwise, frequent false prompts that ask users to quarantine benign programs will add to the frustration of normal users and they might be tempted to turn off the detection system completely.

Low Detection Rate. In order to ensure a nearly zero false alarm rate, dynamic detection techniques typically increase the threshold of confidence on the basis of which a program is declared as malware; as a result, detection rate is degraded that might leave a user vulnerable to attacks.

Robustness to Evasion. Crafty malware writers can attempt to evade the malicious application detection systems. Evasion is possible when malicious applications can reproduce a feature set that is similar to the feature set produced by legitimate applications. This is a significant challenge for security researchers. Most of dynamic analysis techniques tend to detect a process’s behavior by analyzing its behavior in the kernel space of an operating system; therefore, malware try to hide themselves through mimicry of behavior of benign processes. For example, if a dynamic detection scheme monitors pattern of API calls made by the processes, a malware process can try to intersperse its own API calls within sets of benign API calls to evade detection.
2.3.2 Additional Challenges for Smartphone Platforms

After introducing the detection challenges for malware detection techniques, we now focus our attention to challenges that are relevant to smartphones only.

Battery Constraints. Battery is one of the most important resources on smartphones; therefore, any malware detection solution should use it carefully to provide users with enhanced connected times without the need to frequently recharge.

Limited processing power. Detection algorithms should be simple, having relatively small processing complexity, so that they can execute (without degrading a user’s experience) on mobile processors. It is advisable to compromise accuracy for complexity.

Limited Memory. Memory is an expensive resource on a smartphone and detection algorithms should use it in an optimized manner.

Processors architecture Issues. Smartphones typically use processors with low power footprint (like ARM) to conserve battery. It is, therefore, imperative that the detection techniques should ensure that their framework is not tightly coupled with processor specific features.

2.4 Implementation Decisions

In this section, we introduce the design spectrum for malicious applications detection frameworks. A malware detection scheme can plug and play different design options to create a customized detection engine.

2.4.1 Host-based vs Decoupled Security

The malware analysis and detection process can take place at three points: (1) entirely on a smartphone; (2) partially on a smartphone and partially on a remote server; or (3) entirely on a remote server. The process of delegating security away from a smartphone is known as decoupled security.

A designer needs to make an informed choice by critically reviewing merits and demerits of each option. For example, a family of smartphones might be having limited memory, processing and battery resources; therefore, it is prudent to move compute-intensive behavioral analysis and pattern matching computations to a remote server. But sending a program to a remote server can take more time and Internet bandwidth (might be expensive in 3G networks). This can eventually load a central server that results in a degraded user experience. To conclude, a logical compromise is: to do simple analysis on a smartphone and to delegate compute-intensive calculations to a remote server that might be located in a cloud. Jupiter [Guo et al., 2011] provides an environment that augments smartphone environments with the cloud computing.

2.4.2 User space vs Kernel space

The malware detection system can run either in the user space or in the kernel space. For user space implementation, it is necessary that a smartphone operating system provides necessary APIs for collecting the required features (information) about the processes that are running on the system. In comparison, an implementation in the
kernel space provides more flexibility because the behavioral information is collected by hooking the kernel functions and monitoring the kernel structures. In terms of man machine hours, a user space implementation takes less effort compared with the kernel space implementation albeit it is less sophisticated and accurate.

One should remain cognizant of the fact that malware can easily detect a user space program that attaches hook to it; as a result, it refrains from doing malicious activity to avoid detection.

### 2.4.3 Actual System vs Sandboxing/Emulation

The program (under analysis) can either be run on an actual smartphone with normal privileges and restrictions as that of a trusted program, or it can be run in a sandboxed or emulated environment for analysis. Sandboxing provides a confined operating environment with a security policy that restricts the actions that an application can perform. Emulation means imitating the actual operating environment for an application. The purpose of sandboxing/emulation is to analyze the behavior of a program without endangering the actual operating system.

Sandboxing is sometimes implemented by the operating system of a smartphone. The second option is to install third party sandboxing tools for smartphones. The third option is to use the concept of decoupled security and execute the program (under analysis) on a remote server within an emulator. Sandboxing can be done either in user space or in kernel space. Both of them suffer from the same merits and demerits as already discussed for user and kernel space malware detection systems.

The sandbox is configured to intercept and log the changes made to a system by an executing program. The objective is to log a number of features such as system and API calls, changes to a filesystem, resource utilization, network activity, battery drainage and other parameters of a system that define the system’s health and responsiveness.

A program is installed in a sandbox environment. The system maintains a log of changing features during its execution. The programs on smartphones are mostly interactive; therefore, a simulated user input is required for automatic analysis of the sandboxed programs. The programs such as Android’s Monkey\(^1\) tool allows simulating random user events at different intervals. By simulating user events, a program’s behavior in response to the user events is logged. The log of features can be used for dynamic malware detection (during or after execution) by employing techniques like function call monitoring, power utilization, behavioral analysis etc.

### 2.4.4 Static Detection vs Dynamic Detection vs Hybrid Detection

Analysis to detect malicious applications can be performed in two different ways [Chandramohan and Tan, 2012]. Static detection is performed before a file is executed, using the features set extracted from an executable file. Dynamic detection, on the other hand, is performed during or after the execution of a program (usually in an isolated environment). Static detection is relatively quicker and inexpensive, and it doesn’t require execution of the application; but, it is easier to evade. Dynamic detection systems degrades the performance of a system because of the associated

---

\(^1\)Monkey tool can be executed on an Android OS by the command: `adb shell monkey`
processing overheads. A detection system for malicious applications can be either static or dynamic or a hybrid of both approaches. A hybrid system combines the benefit of early detection of static analysis with the robustness benefit of dynamic systems against evasion attempts.

### 2.4.5 One Class (Anomaly Detection) vs Two Class Classification

The basic purpose of a detection system is to classify a given program as benign or malicious; therefore, detection can be treated as a two class problem. But, using anomaly detection systems, it can be treated as one class – the system learns the normal behavior of a program and any deviation, measured with information-theoretic distance measures, from normal is classified as malicious (also known as outlier detection). The anomaly detection can be based on statistical measures (e.g. mean, variance), distance measures (e.g. nearest neighbor, clustering) or models/profiles.

The anomaly/outlier detection is more suitable when it is difficult to get adequate number of unique samples of one class (e.g. malicious). However, for doing anomaly detection, the complete picture of one class behavior is needed to train a classifier (a daunting task in itself). If large samples of both classes are readily available (e.g. Android, Symbian OS), two class classification makes more sense. On the other hand, anomaly detection is a more logical approach for operating systems with limited number of malicious applications (e.g. iOS, Windows Mobile).

To summarize, we have introduced malicious applications types, common infection vectors, design challenges and implementation decisions to be taken in building intelligent malicious applications detection frameworks. Armed with this knowledge, we now focus on a generic framework to design and develop detection systems for malicious applications on smartphones.

### 2.5 Generic Malicious Applications Detection Framework for Mobile Computing Devices

In this section, we introduce a hybrid security framework for detection of malicious applications (causing security threats and privacy leaks) on smartphones. This framework provides a cohesive view of two paradigms (i.e. static and dynamic detection). Moreover, it serves as a common abstract framework through which different instances (by selecting suitable modules) can be instantiated. As a result, it becomes a blueprint for designing future security solutions for detecting malicious applications. Last but not least, it acts as a reference framework for comparing implementations of different solutions, their merits, demerits and performance.

Static and dynamic analysis techniques usually share some common archetypes. For example, the first step is to extract the features that are relevant to a program or process. Then, statistical and information-theoretic measure analysis is performed to select features having the ability to discriminate between two different types of processes. The final step is to make a decision by using classification and prediction algorithms. Despite of having these similarities, significant differences do exist between both analysis paradigms. The static techniques are usually employed before execution of a process (i.e. first-line-of-defense) while dynamic techniques
Figure 2.1: Proposed generic malicious applications (security threats & privacy leaks) detection framework for smartphones
are employed either at run-time or after the process has executed (i.e. second-line-of-defense); therefore, combining both of them in a framework can help provide a two-layer defense against emerging zero-day malware.

The proposed hybrid framework consists of two layers: (1) Static Analysis and Detection Layer (SADL); and (2) Dynamic Analysis and Detection Layer (DADL). The prominent modules of SADL are: (i) file scanning agent, (ii) features processing agent, and (iii) classification engine. On the other hand, the top level modules of Dynamic Analysis and Detection Layer (DADL) are: (i) process and system monitor, (ii) spatio-temporal processing agent, and (iii) classification engine.

The implementation of the framework operations can be done either in the kernel space or the user space or both. (Generally speaking, it is preferred to implement the dynamic layer in the kernel space.) Figure 2.1 summarizes the flow, interaction and characteristics of different modules. The prominent features, their characteristics and functions of layers along with their submodules – provided in the following subsections – are based on common solutions proposed for both static and dynamic analysis of smartphone applications.

### 2.5.1 Static Analysis and Detection Layer (SADL)

Before the program is executed, a binary executable is processed by SADL which performs static analysis to detect security threats or leak of private information. This layer further consists of the following submodules.

#### 2.5.1.1 File Scanning Agent

The file scanning engine analyzes the executable and extracts basic features from it. Static analysis based security solutions on smartphones use features that are extracted from source code and binary executables. Some examples of such features are executable’s structural features [Shahzad et al., 2012], system calls and library calls in binaries [Schmidt et al., 2009a] [Schmidt et al., 2009b], n-gram of assembly instructions [Masud et al., 2008], malicious patterns analysis in disassembled code [Blasing et al., 2010] and static taint based features [Gibler et al., 2012] etc. Different solutions use different nomenclature for such features and components. Therefore, we have given the generic name ‘file scanning agent’ to this module in the static analysis and detection layer of our proposed hybrid framework.

#### 2.5.1.2 Feature Processing Agent

The feature processing agent is the second major module within SADL. It uses preprocessing filters to remove the features that are not useful during classification i.e. the features with zero or very low classification potential. It makes sense to eliminate the features of least predictive significance or combine them with other similar types of attributes; as a result, the dimensionality of input attributes space is reduced. Consequently, the detection accuracy is not only increased but processing overheads are also reduced. The common practice is to perform dimensionality reduction by utilizing information-theoretic measures such as information gain or gain ratio measurements, Fisher Score, Chi-squared distribution [Shabtai et al., 2012], principal components selection and discrete Haar/Wavelet transform [Shahzad et al., 2012]
2. A Survey on Malicious Applications Analysis and Detection for Smartphones

The dimensionality reduction also helps in reducing training and testing time requirements of classifiers.

2.5.1.3 Classification Agent

The primary goal of classification engine is to accurately classify a given feature set as belonging to either benign or malicious class of executables. Most of static detection systems on smartphones use machine learning classifiers – decision trees, inductive rule learners [Shahzad et al., 2012], support vector machines [Bose et al., 2008], bayesian nets [Sanz et al., 2012], neural networks [Barrera et al., 2010] and clustering algorithms [Enck et al., 2010] etc.

2.5.2 Dynamic Analysis and Detection Layer (DADL)

To classify the program during (or after) execution by monitoring its runtime behavior, the Dynamic analysis and detection layer (DADL) is utilized. This layer consists of the following modules.

2.5.2.1 Process and System Monitor

Security solutions, employing dynamic techniques, extract features based on the state of an operating system, runtime behavior of processes and their performance logs. The Process and System Monitor submodule is responsible for extracting these features at runtime during specific time windows. The common feature sets are: process control blocks of executing processes in the kernel [Shahzad et al., 2012], system call and library call logs of processes [Grace et al., 2010], in-execution taint features of programs [Enck et al., 2010], battery utilization based anomalies [Liu et al., 2009], operating system event logs (free RAM, user inactivity, sent SMS count etc.) [Dini et al., 2012][Schmidt et al., 2009].

2.5.2.2 Spatio-temporal Processing Agent (STPA)

In dynamic detection, it is important to estimate and predict the behavior of parameters which change with time and that can be used for malicious application detection. Spatio-temporal Processing Agent is responsible for time series analysis to select relevant features online (while execution) and offline (after the execution of processes is finished) on smartphones. The time series analysis employs statistical techniques such as the moving mean, variance, divergence, time series model building, estimation, and forecasting to compute temporal derivatives of raw features [Shahzad et al., 2012]. Dynamic taint analysis [Enck et al., 2010] is another methodology that is used for realtime or in-execution malware detection. Spatio-temporal Processing Agent also uses the pre-processing filters to eliminate the features or attributes that don’t play a vital role in the classification process.

2.5.2.3 Classification and Prediction Agent

The basic aim of Classification and Prediction Engine is to accurately classify a given feature set as belonging to either benign or malicious class of executables on smartphones. Most of dynamic and behavioral analysis based malware detection techniques use learning classifiers for one or two class classification e.g. decision
trees, inductive rule learners [Shahzad et al., 2012], support vector machines [Bose et al., 2008], Bayesian nets, neural networks and clustering algorithms [Dini et al., 2012][Enck et al., 2010] etc. Moreover, outlier or anomaly detection is also employed on smartphones [Shabtai et al., 2012][Schmidt et al., 2009].

After introducing the generic framework, we now describe the common malicious applications detection techniques for smartphone platforms. This will help to build a basic understanding of common methodologies, employed by security researchers, to predict and classify smartphone applications as benign or malicious. We have categorized the techniques into static and dynamic analysis paradigms to maintain the flow of discussion.

2.6 Static Analysis Techniques for Detection of Security Threats and Privacy Leaks on Smartphones

Analyzing structure, source code, functions and system calls in a binary executable (or in the source code if available) – residing on a disk or preparing to be launched (without executing it) – is called static analysis. Static analysis techniques are mostly applied on different forms and representations of a program code or executable. In case of source code availability, the static analysis techniques help analysts in finding memory leaks and corruption faults. They also play an important role in quality assurance and correctness of models for different systems. If the source code of an executable is not available, static analysis tools use its binary instructions, disassembled assembly code and the information in structural headers for classification. This section deals with various techniques and approaches that have been applied to perform such static analysis.

2.6.1 Malicious Code Pattern Detection

A simple and fast way to detect a malware (especially the ones derived from previously known malware) is similar to the signature detection schemes. The mobile application’s source code can be searched for existence of malicious patterns of code (or other resources). It is performed at the time of installing applications; therefore, it doesn’t result in degrading a user’s experience. Obviously, this methodology can be evaded using code obfuscation and polymorphic techniques to create malware.

In most cases, the source code of the mobile applications is not available; therefore, the general steps performed in this scheme are: (1) the application installer packager is decompressed to extract the files (on some mobile platforms (e.g. iOS), the binaries are not only signed but are also encrypted, so decryption needs to take place.); (2) after decryption, binary might need to be unpacked to obtain plain executable; (3) the executable is then decompiled/disassembled to get the assembly listing of the binary (In case of applications compiled as intermediate byte code (e.g. JAVA classes), inexpensive decompile techniques exist to get the original java source.); (4) the observed malicious patterns are stored in a database; and (5) the final step is to search for malicious patterns – a code block, a specific API call or pattern of calls, a combination of permissions, call to the native runtime environment, attempts to bypass the permissions, or attempts to use services or provisions that
would quickly deplete the battery – in the executable. A match triggers a malware alarm.

2.6.2 Static Function Call Analysis

Some researchers use static analysis of function calls in a program to differentiate between benign and malicious processes on smartphones (functions are reusable blocks of code that perform a specific task.). In this methodology, a programme that uses certain function calls is classified as malware. Any good disassembler has a built-in capability to list function calls, existing in the code of a program, and the resulting assembly code (obtained from disassembly of a binary program) to build a function call flow trace. In some binary formats (like ELF) relocation and symbol tables contain the information about function calls. The static function call analysis is done on the pattern of function class flows of a program. It uses statistical distance measures to match the pattern with malware or benign models.

2.6.3 Static Permissions Leak Detection

Most mobile operating systems allow/disallow use of their resources by defining and applying a permission or capability model. Each application can only access the permissions that it has specifically requested at the time of its installation. These permissions are also confirmed by a user at installation time (or first time use) depending on the policy of an operating system.

Two interacting applications can by pass the permissions model by invoking indirectly the services of another application. Assume that an application A has the permission to access Internet. If another application B can invoke A, it is possible for B to access Internet indirectly without explicitly getting the permission for it. This phenomenon is known as Permission/Capability Leak. If a privileged application – having permissions to perform a privileged action – exposes an interface for invoking privileged actions, it can help un-privileged applications to invoke privileged actions indirectly.

Interfaces can be defined in a number of ways. For example, Android uses the mechanism of Intents, and the iOS allows applications to register as URI handlers. Different applications from the same developer might use the same identifier. As a consequence, applications from the same developers can use a union of the permissions of individual applications. This permits applications to do actions for which permission is not sought and this leads to implicit permission/capability leak.

Possible control flows of a program (as mentioned before) are created using a control flow graph. Indirect control flows (like threads) also need to be catered. Moreover, a control flow for each entry point (in case of multiple entry points) needs to be created and they need to be merged in a single graph. A leak can be detected by finding a feasible path between an interface and the privileged action. If such a path exists, a permission/capability leak has occurred.
2.7 Dynamic Analysis Techniques for Detection of Security Threats and Privacy Leaks on Smartphones

Dynamic analysis techniques monitor behavior patterns in runtime execution traces to detect security threats and privacy leaks. For this purpose, they analyze the information available in process control blocks of processes, function and system calls – their functionality, call sequences and parameters, taint analysis of executing instructions of programs, performance benchmark counters and parameters of OS. Dynamic techniques are relatively more resilient to evasion techniques – code obfuscation, polymorphism and metamorphism – that are successful against static analysis techniques. Dynamic techniques come in two flavors: (1) post-execution – off-line analysis performed on dynamically produced datasets (system calls, taint data); and (2) In-execution – online analysis in realtime to detect malicious programs during their execution to protect the OS and other user programs from their malicious activities. This section summarizes various techniques and approaches that have been used to perform post-execution and in-execution dynamic analysis.

2.7.1 Information Flow Tracking

A smartphone user would like to protect her/his privacy sensitive information or data that includes user’s contacts, messages, device ID/phone number information etc. Moreover, modern smartphones come with several local monitors – GPS, accelerometer, camera and microphone – that can provide useful information about the location of a user to advertisers. But sharing such information with advertisers without the explicit consent of a user is an obvious breach of her/his privacy but also a violation of her/his trust. This type of sensitive data is tracked by using the concept of tainting: mark/label the sensitive data and then track its usage by different applications.

The taint labels helps in tracking different transformations that are applied to the original taint data. The entry point of taint data into an application is termed as a taint source. Similarly, a taint sink is the point where taint data leaves the application. APIs for accessing contact lists and GPS information are examples of taint sources and network interface APIs – used to transmit taint data over the network – are examples of taint sinks. A taint sink can be programmed to filter tainted data to ensure privacy of a user. Using taint labels, a sink can detect transformed sensitive data provided taint labels are properly propagated from the source to the sink. In case of “direct data propagation” – direct assignments and string and arithmetic operations – taint labels need to be assigned to track the information flow. In “indirect data propagation”, data can be transformed using the address mapping of a known table. An application might map each character in the tainted data to an index of a table (containing only unique values) and then use the index as an address; as a consequence, this indirect memory address of the tainted information needs to be propagated as well.

In case of control flow (if/else), keeping a track of taint labels becomes a daunting task because the information transformation may span over multiple instructions. For example taint data could be tested against different values and its new copy be...
created without resorting to “direct data” or “address” mapping. In implicit flow, the values are indirectly assigned with the help of branches that are not executed; as a result, tracking becomes difficult.

### 2.7.2 Dynamic Function Call Tracing

Dynamically monitoring function calls of a program has been used by the majority of researchers for malware detection on smartphones (and desktop as well). A security driven analysis of invoked function calls of a process provides useful information about the intent of its writer. The main difference from static function analysis is that these techniques analyze the order in which the function calls are made at runtime and generate related function “call flow graph”. Classification is generally performed by comparing the call flow graph of a process with that of benign and malware. They employ statistical distance measures to label the executing process as benign or malware.

To enable this, a program needs to be hooked to log a limited number of invoked function calls – using interfaces defined by the operating system – that are useful (like Software Development Kit (SDK) APIs and system calls). For example, Android applications use Java SDK APIs that allow for easy interfacing with the device.

### 2.7.3 Runtime Permissions Leak Detection

As mentioned before, modern smartphone operating systems use permission models to allow applications to do operations. Kindly recall permissions can be leaked when an unwitting (or possibly colluding) application that has a permission, exposes an interface that allows another application (having no permission) to request a privileged action on its behalf leading to privilege escalation attacks. Moreover, it is also possible for two colluding applications, with different sets of permissions, to share data with each other through covert channels – extending their permission set to the union of both applications’ permission set. These covert channels can be established by applications through shared resources such as contacts database, or observable (and mutable) properties of the system resources (such as volume level, brightness level, etc.).

To detect a permission leak attack at runtime, the interprocess communication needs to be monitored. Specifically, the standard mechanisms provided by smartphone operating system (such as Intents on Android) need to be restricted based on a security policy. An application $A$ that has access to some private data of a user but has no permission to access the network, should not be allowed to communicate to an application $B$ that has permission to access the network.

Even when the standard interprocess communication channels are monitored and restricted, it may be possible for an application $A$ (without network access) to communicate with application $B$ (with network access) through covert channels that can be restricted based on a security policy. For example, if an application $A$ has made a new entry in the address book, an application $B$ should not be allowed to see that specific change.

The biggest challenge in this approach is to create a comprehensive security policy and to reduce the number of false positives.
2.7.4 Misbehavior analysis using power utilization patterns

Another misbehavior detection approach is to monitor and analyze the power consumption of applications running on battery powered smartphones. Typical applications have a specific battery depletion pattern and the basic assumption of this approach is that malicious applications would use a non-standard battery utilization model. The challenge is to model battery utilization behavior of benign applications accurately. The challenges and emerging issues of power based malware detection (presented in [Kim et al., 2008] and [Liu et al., 2009]) are: (1) accurately modeling the battery usage patterns according to a user’s behavioral patterns on a smartphone; (2) monitoring the battery power usage in realtime is a difficult task because the precision of power measurement APIs varies on different smartphones; (3) frequent queries about battery’s remaining capacity also load the CPU and discharge the battery; and (4) polymorphic variants of malware need to be detected to keep the false positive and false negative rates at an acceptably low level.

Different methodologies have been proposed to accurately measure and model power consumption. The well known are two: (1) external measurement, and (2) on-device battery status APIs. The external or physical measurement of power consumption requires external sensors and probes. As power is a product of voltage and current drawn over a unit time; therefore, both of them need to be monitored. Mostly battery maintains the voltage within an acceptable error range; as a result, only the current drawn needs to be sampled over time. One way is to do the measurement directly by intercepting a smartphone’s battery circuit. An indirect way is to measure the magnetic field, produced by the current, using the well known phenomenon of Hall effect. In order to generate the power profile of an application, sampling by the power monitor needs to be synchronized with the real on-device activity. Most mobile operating systems also provide APIs for reading the current battery status. Using a timer, the battery status can be sampled to measure the power consumption of an activity or application. The problem, however, is a significant loss of accuracy. The results are reported as battery segments or remaining battery in percentage, with a granularity between 5 or 6 segments to 100 segments. Latest smartphones provide 1% resolution for battery status and it is only marginally acceptable.

A typical classification paradigm is: to compare the power consumption profile of a phone with that of the one logged during normal operations. If no malware is running, the power consumption profile should be similar to that of the logged one; otherwise, in case of a discrepancy an alarm about a malicious activity is generated.

2.7.5 System Performance/Behavior based Anomaly Detection

Another dynamic approach is to monitor a system’s performance or behavior, and identify the anomaly in this behavior to detect malicious applications. The underlying assumption is: a system’s performance parameters differ in a significant manner because of presence or absence of a malware activity.

In this technique, the goal is accomplished through periodic monitoring of different activities (including but not limited to): message activity, telephony activity, filesystem activity, CPU utilization, memory consumption, network activity, battery drainage, processes and threads, and other parameters relating to a system’s health.
and responsiveness. These metrics are measured for normal user activities or in the presence of malware activity or both. The profile of a system behavior for normal and malicious activities is termed as benign or malicious profile respectively.

The common mechanism to present such profiles is through mapping points in a multi-dimensional space, a set of boolean or comparison rules, and probabilities etc. The profiles can be updated when a new malware is detected. The classification is done by applying data mining techniques on the profile. Typical machine learning algorithms used by researchers for malicious behavior detection on smartphones are: Bayesian Networks, K-Nearest Neighbors, Random Forest, Artificial Immune System, Radial Basis Function and Self-Organizing Maps.

The true merit of this technique is the ability to detect packed/encrypted malware. No unpacking/decryption is needed because the impact of an executing process on a system’s behavior is observed. Moreover, it is possible to detect zero-day (unknown) malware that do the same malicious (as that of known malware) activity but by using different instructions, steps and processes.

The major challenge in this technique is: selecting thresholds for different performance parameters that discriminate normal and malicious behavior. Moreover, for anomaly detection, the complete picture of normal or anomalous behavior needs to be defined and then used in training. A benign process that is different from the processes used in training will be definitely classified as malware (a false positive); therefore, achieving low false positives is not an easy task.

After discussing the common static and dynamic malware detection techniques for smartphones, we now focus our attention to the existing tools and frameworks that utilize them. We restrict our discussion to the major tools and frameworks only, proposed in the recent years, with an aim to choose the representative tools for the above-mentioned techniques.

2.8 Security and Privacy Analysis & Detection Tools for Smartphones

In this section, a critical review of tools and frameworks for analysis and detection of malicious executables and programs on different smartphone operating systems is presented. A short description of different tools along with their analysis and detection methodology is discussed. Specifically we focus on classification efficiency, processing overheads, scalability issues, robustness and resilience against evasion techniques. Finally, we provide relative merits and demerits of a technique compared with others.

2.8.1 Woodpecker

Woodpecker [Grace et al., 2012b] is a security tool that exposes privilege escalation attacks in Android based smartphone applications. It is capable of detecting both explicit and implicit permission leaks. The authors have defined explicit permission leaks as “use of the public interface of a privileged application by a non-privileged application to circumvent the operating system’s permission model”. On the other hand, implicit permission leaks happen when applications share permissions by using shared developer keys.
Woodpecker detects possible permission leaks in the installed (or to be installed) third party applications as follows. First, the Dalvik code is extracted from an application’s executable and it is disassembled. Then, Woodpecker generates control flow graphs from the byte code. Since the application may have several entry points, control flow graphs are generated from each entry point. Woodpecker also takes care of indirect flows (such as thread runnables in Java) to maintain control flow connections.

Using control flow graphs, the capability leak is detected through identification of feasible paths that contain dangerous calls (permission dependent functions). For each entry point, the control flow graph is traversed and the feasibility of each path is computed. If a feasible path passes through any dangerous call, it is flagged as a capability leak. For implicit leak detection, Woodpecker crawls the manifest files of all applications and computes union of permissions for applications with shared identifiers (the applications developed by the same developer). Then, woodpecker uses the control flow graphs to detect if such a permission leak is being exploited by any application. In the case of implicit leaks, Woodpecker generates control flow graphs from entry points and looks at their initialization routines as well.

The authors tested Woodpecker tool on 8 different phones from 4 different vendors. The phones were selected to ensure diversity of tested applications on the Android platform. The authors selected a set of 13 privileged permissions which include: getting a user’s location, making calls or sending messages, deleting user’s data and recording user’s conversations. Their analysis shows that 11 of these 13 permissions were explicitly leaked on the chosen phones. Some phones leaked nearly 8 of these permissions. This means that a third party application can perform a privileged operation by using a leaked permission without the need to ask for a permission from the user or the system. The authors claim that although a large number of possible paths with capability leaks are returned by the Woodpecker, the path feasibility calculation in the Woodpecker tool removes the false trails. The manual verification of the results confirm zero false alarms. The false negatives (in this case) are not reported, which is understandable as the authors don’t have knowledge of all the permission leaks. However, the authors could have generated a test suite with different permission leaks, and then tested the tool on it to report any false negatives. Since the tool is designed to be used offline; therefore, the processing time of around 1 hour for each system image (with typically 100 to 150 applications) seems reasonable. If this tool is to be used online at the time of installing applications, the processing overhead needs to be significantly improved.

2.8.2 Static Function Call Analysis for Collaborative Malware Detection on Android OS

The authors in [Schmidt et al., 2009a] have presented a framework for detection of malware on Android OS using static function call analysis of ELF files by using supervised data mining algorithms for classification. A framework is presented for collaborative detection of malware in ad hoc mobile networks. In a simulation environment, the collaborative framework produces significantly better results.

The framework has three components: On-device analysis, Collaboration and Remote analysis. The features are extracted through on device analysis. The data extraction is done at the OS level because the Java framework doesn’t provide re-
quired APIs. A custom written tool *Interconnect Daemon* monitors the filesystem and operating system events. This daemon is responsible for identifying ELF executables on the system. It parses the executables, using the *readelf* tool, and creates a list of reference functions for each running executable on the system.

The list of functions used by an ELF executable are divided into six different attribute sets based on the type of a function (dynamic, relocation and a set of both) and its presence in malware and benign executables (mutually present, set of all functions). The machine learning classifiers – Prism, PART and Nearest Neighbor Algorithms (KNN) – are trained and tested on the dataset of features extracted from both benign and malware executables.

The features are analyzed using *WEKA* tool [Witten and Frank, 2002]. Approximately 100 benign ELF executables present in Android /bin directory, and 240 ELF malware executables found on Internet have been used for training and testing of classifiers. Using 10 fold cross validation, the classifiers are trained and tested on the features extracted from benign and malware executables in a dataset. The results show that Prism achieves zero false positives but has a low detection rate (approximately 70%) and also uses a large set of classification rules. In comparison, PART produces minimal rules set and has a detection rate of over 99%. However, it produces 12-16% false positives. The learning and classification can be performed on a remote server to reduce the processing load on a smartphone. However, the authors have not reported any memory and processing overheads on the system.

To reduce the false negatives, the authors have created a collaborative environment (an ad hoc network) where a node can request its neighboring nodes to help it in classifying malware. Using a threshold of uncertainty, the results returned from the neighboring nodes can be used for properly classifying malware. They have simulated an ad hoc mobile network to prove the claim that collaborative nodes help in reducing false positives. The authors also demonstrate that frequently collaborating nodes reduce the number of infected nodes in an ad hoc network; however, such collaborations lead to a fast depletion rate of the battery of a smartphone.

### 2.8.3 Static Function Call Analysis using Centroid - Symbian

A tool for malware detection through static function call analysis on smartphones has been presented in [Schmidt et al., 2009b]. The framework has been implemented and tested on the Symbian OS. The tool classifies the applications into benign and malware on the basis of a clustering algorithm called *Centroid machine*.

The framework consists of two parts: (1) feature extractor and (2) centroid machine. The list of functions in an executable defines the features’ set. The list of functions is statically extracted using IDA Pro\(^1\). In some cases, an executable might have to be unpacked before extracting features from it. Unpacking is done by using the UnSIS\(^2\) tool on the Symbian platform. After unpacking and feature extraction, feature selection is performed by using statistical techniques. This helps in reducing the number of attributes and increasing the classification accuracy. A small set of attributes help in reducing the memory and processing overhead of the detection technique. The labeled training dataset is divided into the clusters of benign and malware.

\(^1\)[http://www.hex-rays.com/idapro/]
malware datasets by using the centroid machine. The classification is performed by measuring the ratio of distance of a given application’s attributes from the centroid of benign and malware clusters respectively. The application is classified as benign if its distance ratio – within a threshold that is adaptively determined – is smaller from the benign cluster compared with the malware cluster.

To test the framework, the authors collected 33 malware programs available on the Symbian OS and 49 benign popular applications available on the Internet. The experiments highlight the impact of features reduction on the detection accuracy. Moreover, the authors have also compared the performance of centroid machine with Naive Bayes and Binary Support Vector Machine (SVM) algorithms. For statistical significance, the results have been averaged on 1000 runs, and 10-fold cross validation is used for training and testing.

The results show that the detection accuracy is only marginally affected by reducing the number of attributes from 3620 attributes (accuracy: 98.7%) to only 14 attributes (accuracy: 96.5%). For 14 attributes, the Naive Bayes and Binary SVM achieve accuracy of 90.2% and 91.9% respectively. The authors do claim that the centroid algorithm is efficient and light-weight (making it suitable for running on resource constrained smartphones) but they have not provided its empirical evidence by reporting memory and processing overheads.

2.8.4 PiOS

Another static analysis based tool PiOS is proposed by [Egele et al., 2011] on iOS platform to detect information leaks. It tracks information flow through static analysis of Mach-o executables. In this approach, the tool generates control flow graphs (CFG) from binary executables. In Objective-C, the function calls are bound to the function instances through the Objective-C runtime library; as a result, the function calls are replaced by messages calls (msgSend). To detect a correct function call, the authors have built a hierarchical structure to identify the derived and relevant base classes. Afterwards backward slicing is used to track the input parameters and their type to a dispatch function. In this way, the authentic targets of function calls are determined as a pre-requisite of CFG and it leads to successfully constructing CFG.

In the next phase, the framework performs the reachable analysis to determine the existence of paths (hierarchical function calls), which provide connectivity of the source of the sensitive data to their sinks – modules that provide connectivity and communication. This analysis on CFG determines the information leakage from iPhone devices to the third-parties or hosts. In the final phase, the data flow analysis is performed to validate it. Finally, PiOS generates the list of source-sink pairs for the analyzed information flows (the pairs represent different private information leakage scenarios). The tool allows a user to manually inspect the list of pairs that are linked even though no information flow is seen between them.

The authors have analyzed 1400 iPhone applications using PiOS. They report that other than few ‘bad apples’, most of the test applications didn’t leak sensitive information. It is interesting to note that more than 50% applications secretly leaked the identity of the mobile device. This leak, coupled with profiling a user’s smartphone usage pattern, has the potential of a privacy leak.
2.8.5 SmartDroid

Smartphone applications are typically highly interactive, which means that a malware can hide its activity from automatic malware detection tools by hiding the trigger for the malicious activity in complex user interactions. *SmartDroid* [Zheng et al., 2012] is a framework that attempts to solve this problem by finding such user-interface (UI) interactions through a hybrid (static and dynamic) analysis of the application and simulating the triggering interactions. The static analysis module i.e. *static path selector (SPS)* determines the possible activity switches and function calls paths. It performs this analysis by disassembling the application and constructing function call graphs (FCG) and activity call graphs (ACG). All indirect and event-driven APIs are also included in the construction of CFG. The ACG is constructed through analysis of explicit and implicit Intent constructor calls. Dynamic analysis is performed to match the UI-elements with their related UI-event functions. A dynamic UI Trigger is created by modifying the Android framework and building a modified emulator. The UI-Interaction Simulator helps in performing dynamic analysis in an automated manner through traversal of the UI tree and simulating interactions with the UI-elements in activities.

The prototype of *SmartDroid* framework is evaluated using 19 malicious applications, belonging to seven different malware families. The authors report that SmartDroid is able to efficiently unveil the simple indirect UI-based trigger conditions. However, some of the complex indirect conditions (data based UI elements creation) for trigger are missed. The static analysis mean time is reported in the range of 5-16 seconds, while the dynamic analysis mean time is typically about half a minute per path. This framework is suitable only for offline analysis or decoupled analysis of the potentially malicious applications because it requires changes in the Android framework and relatively large analysis overhead. Moreover, the framework has only been tested on a very small and selected malware dataset.

2.8.6 Multi-Level Anomaly Detector for Android Malware (MADAM)

The authors of [Dini et al., 2012] have proposed a multi-level anomaly detection framework that detects malicious application through dynamic analysis in realtime using machine learning classifiers. As the framework uses anomaly detection instead of two class classification, it is supposed to be capable of detecting zero-day (previously unseen) malicious applications. The framework monitors the smartphone on two different levels: the kernel space and the user space. The system calls made by the smartphone applications are intercepted and logged in the kernel space using a kernel module. The list of running processes, the memory usage and the CPU utilization are also monitored in the kernel space. Moreover, a user’s state (active or idle), key-strokes, called numbers, SMS, Bluetooth and WLAN activity is analyzed and logged in the user space. This multilevel view of the system events helps in a wider range of features (used for monitoring) and provides a correlated view of the events occurring at different levels.

The authors have employed k-nearest neighbors (KNN) (with k=1) algorithm for classification of the collected feature set. A dataset of 10 malicious and 50 legitimate applications on Android is used for evaluation purpose. The authors have divided classification process into three phases: (1) training phase, (2) learning
phase, and the (3) operative phase. The framework is claimed to be doing anomaly detection but its classifier is trained using feature vectors of a normal user behavior and synthetically created malicious behaviors. In the learning phase, the classifier is trained for a user-specific behavior to estimate the false alarm rate (a decrease in FAR is reported from 26% to 0.1% as the analysis time increases from 10 to 240 mins respectively). The authors show that the framework can adapt by gradually adding new elements in the training set at run-time. Finally, in the operative phase, a classifier does anomaly detection on short-term (1 sec) and long-term (60 sec) timing windows. An average detection rate of 93% along with an average false positive rate of 5% is reported. The performance overheads are: 3% memory utilization, 7% CPU overhead and 5% battery depletion. The major shortcomings of the framework are: (1) high false alarm rate; and (2) evaluation on limited number of real malicious applications. The authors claim that the framework is capable of detecting zero-day malware.

2.8.7 Virus Meter - Battery Utilization patterns - Symbian

A misbehavior analysis and detection tool Virus Meter, which uses battery utilization patterns of applications, is presented and demonstrated for Symbian OS on Nokia 5500 smartphone [Liu et al., 2009]. Its core principle is to detect energy hungry applications [Kim et al., 2008]. The authors have profiled energy utilization of applications based on a user’s activity (the duration of voice calls, frequency of sending/receiving text messages, usage & processing of documents, a system’s idle state, entertainment and other activity benchmarks), and a system’s performance/benchmark parameters – signal’s strength & weakness and the network’s activity and its state & conditions. It is argued that these conditions significantly effect the battery utilization behavior: for example a long voice call, frequently sending & receiving SMS and low signal strength results in more battery depletion. Three power calculation functions are approximated by using machine learning algorithms (linear regression, neural networks and decision trees) to compute the power consumption between two subsequent power measurements.

A state machine is defined to model a user’s behavior that is previously defined. This state machine consists of a sequence of steps: (1) the power consumption monitoring application is installed on a clean (no malware) smartphone OS; (2) a known process, whose power consumption is to be measured, is launched; (3) the relevant events along with their characteristics are identified and recorded; (4) an association and correlation of events with the launched process is computed and relevant features of the events are recorded; and (5) finally, steps 1 to 4 are iterated to identify the sequence of common events.

Using the power models and state machine events, the difference between predicted power and the measured power is calculated. If abnormal patterns are observed between the two, the application is declared as malicious. Virus Meter uses linear regression model to detect misbehaving applications in realtime. The linear regression model reduces the processing overheads but realtime prediction might not be accurate due to oscillatory behavior of electro-chemical batteries. This might increase the false positive rate in misbehavior detection. The machine learning algorithms – decision tree and neural networks – provide better results and reduce the probability of false positives because they give a decision over time series collected
data instead of individual instances. The machine learning algorithms consume relatively more processing and battery power; therefore, they are only used in the charging mode.

The framework is evaluated using known Symbian malware FlexiSpy and different variants of Cabir. FlexiSpy is basically a spyware, designed to do silent calls, interception of calls and SMS forwarding etc. Cabir uses bluetooth functionality to spread itself. The misbehavior (min-max) detection rate of the framework on detecting silent calls, calls interception and SMS forwarding is 85-93%, 66-90% and 89-98% respectively in both realtime and charging modes. On the other hand, Cabir’s variants detection rate is 89-93%. Moreover, the false positive rate of the framework is 4-22% using the above mentioned predictor and classifiers. Furthermore, the processing overhead of the framework is 1.5% in terms of power consumption using linear regression. To conclude, the virus meter is not suitable for realtime deployment because of its high false alarm rate and inconsistent behavior of electro-chemical batteries (due to power fluctuations).

2.8.8 Energy-Greedy Anomalies & Malware detection - Windows Mobile

This framework uses signatures, derived from the power utilization history, to detect energy greedy malicious applications [Kim et al., 2008]. It is developed for HP iPAQ smartphone on a Windows Mobile operating system. The framework consists of two major components: (1) a power monitoring module and (2) a data analyzer module. The power monitoring module collects, analyzes and maintains power consumption – calculated by the product of instantaneous current and voltage over a fixed time – of different applications running on the phone. The power is precisely calculated, by using an oscilloscope with a hall effect probe, which measures the current drawn by the phone (since the voltage is constant; therefore, the power is directly proportional to the current drawn).

The malware can learn the sampling pattern and accordingly change their execution behavior; therefore, the authors use two different power calculation methods. In the first method, power samples are taken after a fixed period of time but the starting and ending points are randomly chosen – making the measurement interval random. In the second method, even the frequency of collecting samples is made random as well. The monitoring module also computes and stores a mean value of different power levels for each state of the smartphone e.g. on, backlight off, screen off and on, backlight on and screen on etc. Once the power consumption approaches near to the threshold level, the power monitor raises an alert and begins to store the energy utilization record. To achieve better accuracy, a higher sampling rate is preferred but that might lead to more power loss.

The data analyzer component extract patterns from the collected samples and generates signatures. It uses a moving average filter to remove high frequency outliers in the sampled data and it uses a customized compression algorithm to reduce processing overhead of matching the generated signatures with the ones in the signature database. Newly installed or signature-less applications are generally misclassified.

Using the above-mentioned scheme, battery depletion attacks are detected with 100% accuracy – different programs (e.g. WiFi faker and dummy programs etc.) and their combinations are evaluated. The framework is tested using four mobile
worms i.e. Cabir, Lasco, Commwarrior and Mabir (they belong to the same malware family). The authors have shown that the power signature of one malware can be used to detect other unknown (zero-day) malware of the same family. The accuracy to detect one worm varies between 93% and 100%. On the other hand, the detection accuracy of a family for worms – detected with the same signature – varies from 80% to 93% (and 100% only in the case of Mabir worm family). The false positive rate is approximately 2%. The processing overhead of the framework, in terms of battery utilization, is not reported and the robustness of battery based signatures is not analyzed. A crafty attacker can design a malicious application that consumes the same power utilization pattern as that of a benign application; as a consequence, it can successfully masquerade as a legitimate application.

2.8.9 Cloud-based Paranoid - Android

A dynamic and decoupled security solution Paranoid is presented in [Portokalidis et al., 2010]. To avoid the resource extensive implementation of a security framework on a mobile host, the authors have suggested a cloud-based security framework for Android. They record and replicate the minimal instruction traces of Android’s executing processes to a remote server in the cloud. On the server, the collected instruction set of processes’ are replayed in the Android’s emulated environment and a multi-layer forensic analysis is performed to detect malicious processes.

The flow of information among different components in the Paranoid framework is summarized. An instruction tracer records (using ptrace system call) program instructions and stores them into a secure storage, available on a mobile host. To protect the blocks of instruction traces from being tampered by any intruder or malicious software, a message authentication code (in conjunction with the hash key) is attached with every instruction block. To protect the previous blocks in the storage, key rolling approach (hashed MAC code of the previous key) is used to calculate the hash of the new (to be) used key. To avoid any failures (e.g. battery power problems), instruction traces are only synchronized with the cloud server when the smartphone is in the charging mode. To record and replay the complete execution process of a program, the recorded instructions and the input data – provided by the user through hardware – is required. Paranoid doesn’t store the input data with the instruction blocks; therefore, a separate proxy server is implemented on the server side to fetch the data on demand. The instruction blocks and data are transferred from a mobile host to the cloud in a compressed form. An Android emulator is installed on the cloud server to perform the security checks during the replay of a program’s instruction trace. The checks are: (1) code injection and buffer overflow attacks; (2) a signature-based on access scanning of files by the open source antivirus; (3) an emulator’s memory scan for malicious codes; and (4) anomaly detection using the system call traces of the programs.

In the emulation environment of a cloud server, the malware detection accuracy and false positive rates are not reported explicitly. In general, the authors have emphasized the processing, storage, power consumption and replication overheads. They have collected data and process traces from real smartphone users’ (more than 100 smartphones can connect to the cloud and replicate data.). It is reported that, most of the time, a smartphone remains in an idle state or is used for voice calls. The data rates in the idle and busy states are approximately 64-120 B/s and 2
KB/s respectively. The volume of collected data of voice call traces exceed 20 MB. Replicating data from a smartphone to the cloud server can have significant costs in terms of bandwidth hogging and the price charged by 3G operators. A better approach would be to store them in a local memory and use WiFi connections instead. The framework increases the processing load of a smartphone by more than 15% and battery depletion rate by 30%. Both overheads are quite significant.

2.8.10 Crowdroid - System calls based decoupled security

This framework employs dynamic monitoring of system calls of Linux kernel to detect malicious applications on Android. It uses decoupled malware detection approach to reduce processing and power overheads [Burguera et al., 2011]. The authors have developed a smartphone client application – crowdroid – that uses crowd sourcing. It consolidates the log of system calls of different applications, running on multiple smartphones, and uploads the data on a remote server. This framework employs the strace tool for logging systems calls and ftp to upload the data to the remote server. The users share only behavioral data related to the used applications and not their confidential or personal information.

On the server side, the dataset is parsed by a data processing module that uses perl scripts to do behavioral analysis and generates different behavior vectors based on the system calls for each application. These vectors contain the information of accessed files, execution duration of processes, and a count of system calls used by the application programs. Afterwards, the k-mean clustering algorithm is applied for classification. The classification results for every individual application are stored in the database.

The authors have tested the framework using two real malware i.e. PJApps embedded in a steamy window application and HongTouTrojan embedded in a monkey jump application. They obtained benign and infected Android applications from known online repositories. They collected data from 20 users. The data contained 60 system call traces from benign and self synthesized malicious applications (50 benign and 10 malware). A 100% detection accuracy is reported for self-created malware. For realworld malware, authors have used 20 features’ vector of benign and malicious applications. They have reported 100% detection accuracy for PJApps malware and 85% accuracy for HongTouTrojan. The false positive rate is 20% in case of benign-HongTouTrojan clusters. Such a high false alarm rate is not suitable for a real world deployable application. Experiments are needed to show the robustness of their features set against evasion. Moreover, the overhead of the proposed algorithm is not evaluated.

2.8.11 Knowledge-based temporal abstraction - Android

A lightweight, host based intrusion and malware detection security solution to detect unknown (zero-day) malware by monitoring time series measured data and events on Android phones is presented in [Shabtai et al., 2010b]. This framework is adapted form of a knowledge-based temporal abstraction (KBTA) [Shahar, 1997] [Moskovitch and Shahar, 2009] ontology for smartphones. KBTA is based on five different types of parameters: (1) primitive features (i.e. CPU usage, sent or received data packets over network interfaces etc.); (2) abstract features – they are
derived from the primitives (i.e. percentage of CPU’s busy state etc.); (3) the events are a form of raw data based on a user’s (or system) behavior (the number of screen touch events, the number of applications launched and terminated etc.); (4) the context is used to assign meaning to different type of parameters: the CPU state in a user’s busy or idle state etc., and the classification function of parameters changes with respect to their context (they are further classified into different types: state (high or low CPU activity), trend (escalating or decreasing rate of camera activity) and rate (quantified rate of change of a feature value) etc.); and (5) the patterns are derived from parameters, their context and associated events by defining local and global timing constraints.

The framework consists of the following components: (1) feature manager, (2) agent service, (3) processors and (4) a graphical user interface. After a periodic interval, the feature manager module extracts features from different layers of OS. The processor unit is mainly an analysis and detection framework that consists of a machine learning classifier for misbehavior detection. It receives dataset from the feature manager and classifies it after doing relevant processing. Finally, it forwards the results to threat weighting units (TWU) that apply selection or summation algorithms — majority voting or distributed summation etc. The final results are forwarded to an alert service that applies a smoothing filter and min/max thresholds. The agent services module provides the organization and communication services to all components. The graphical user interface is used for configuring agents, warning alerts and visual exploration of the extracted data. The framework uses a fuzzy algorithm that works on a collection of constraints (instead of the classical signature based approach) to detect malware.

Different types of sample malware applications are used to test the framework: a game for information stealing (camera pictures), a tip calculator for denial of service, a malicious application to block outgoing calls, an information leakage (from SD card) application and a contact stealing and transmitting application. The frequency of data logging is selected from four intervals that vary from 2 to 14 seconds. The detection rate of applications (in the best case scenario of 2 sec sampling intervals) is 98-100% and detection time varies in between 5-32 seconds. If the sampling intervals are increased beyond 2 sec, the detection rate and time are only marginally effected. The authors have not reported false alarm rate even though they have intuitively presented scenarios that could lead to false alarms. Moreover, they have also not discussed robustness of their fuzzy algorithm against evasion attempts.

### 2.8.12 Andromaly

It is a host based anomaly detection security system for Android smartphones [Shabtai et al., 2012]. Its architecture is inspired from the knowledge base temporal abstraction framework [Shabtai et al., 2010b]. The authors have used two Android HTC G1 smartphones that have 20 installed benign games. To incorporate behavioral changes, the phones were given to two different users. The authors used 3 features reduction methods – Chi-Square, Fisher Score and Information Gain – to rank the features (they created three datasets with 10, 20 and 30 top ranked features). Later they classified the datasets with K-Means, Logistic regression, Histograms, Decision tree, Bayesian networks and Naive Bayes. The authors preferred light weight classifiers to optimize power consumption.
They have created four different training/testing scenarios: (1) In the first scenario, training set consists of 80% benign and malware samples and the remaining 20% is used for testing; (2) In the second case, they trained on the dataset of 3 malware and 3 benign applications and tested on one malware and one benign application; (3) In the third scenario, the dataset of the first phone is used for training and the system is tested on the dataset of second phone; (4) In the last scenario, the dataset of three benign and malicious applications of one phone are used for training and the system is tested on one malware and one benign application of the second phone. The obtained results for four scenarios are: (1) In scenario 1, the decision tree outperformed other classifiers and provided 99.9-100% accuracy with 0% false positive rate (FPR); (2) In the second scenario, logistic regression provided the best results (86-90% accuracy with 12-11% FPR); (3) For the third and fourth scenario, Naive Bayes achieved 82-88% accuracy with 23-14% FPR and 75-85% accuracy with 29-17% FPR respectively. Andromaly uses 8.5% RAM of a smartphone and its processing puts 3.5-7.5% additional load on the CPU and its detection time is 5 sec. Overall, it degrades performance of smartphone by 10%. A very small set of malicious applications has been used for experiments and a more comprehensive evaluation is needed.

2.8.13 Behavioral Misuse Detection - iPhone

A misuse detection system based on a users’ data logs on iPhone is proposed in [Damopoulos et al., 2012]. The classical machine learning algorithms – random forest, bayesian networks, radial basis function and k-nearest neighbors – are applied to a dataset that contains a user’s voice calls, text messages and Internet browsing history. The data is logged individually (for each individual application) as well as in a multi-modal fashion (combined data from all services).

To collect the dataset from 35 iPhone users, the authors designed a client-server that logs and stores the dataset. The voice calls records consist of the number, flags indicating incoming/outgoing calls, the time stamp and call duration. The features’ set from text messages is derived in a similar fashion. Moreover, the browsing history is maintained by logging the web-link and time stamp parameters. The outcome data analysis is: (1) 66% users used their iPhones for Internet browsing and they hardly visit previously unknown URLs; (2) only 2% text messages were sent and received from new mobile numbers while other 98% were from known family and friend numbers (almost a similar trend is observed for voice calls).

The authors have used two different validation schemes: 10-fold cross validation and 66% data split. Using random forest classifier, the system achieved 99.8% true positive rate (labeled as sensitivity) and 0.3-0.4% false positive rate to detect misuse of calls, text message and browsing services. They claim that their detection rate is 1.2% superior compared with the previous solution [Liu et al., 2009]. The error rate and false negative rates are 1.6% and 0.7% respectively. The aggregated detection time varies in between 1.5 and 6 seconds for both validation schemes. The robustness of the used features set is not discussed by the authors.
### 2.8.14 TaintDroid

TaintDroid [Enck et al., 2010] is an information flow tracking tool for Android smartphones. It attempts to improve the visibility of sensitive data as it flows through the third-party applications and empowers a user to control its use. The tool is capable of tracking multiple sources of privacy sensitive data and provides dynamic taint tracking capability to the system. TaintDroid modifies the virtual execution environment of Android operating system. It applies taints (labels) to the sensitive data, tracks its flow and propagates associated taints. Finally, an alarm is raised if the labeled data leaves the system through a non-trusted third party application.

To efficiently track the information flow, TaintDroid applies labels at four granularity levels in the system: variable-level, method-level, file-level and message-level. The variable level labeling tracks intra application (running in a virtual environment) data flows. To track data flow in API calls, method-level tainting is performed. File-level taints are used to monitor storage and network I/O data flows. Interprocess communication data is tracked through message-level taints.

TaintDroid exploits spatial locality, to efficiently store and retrieve data labels, by storing data and labels adjacent to each other. The storage overhead is minimized by storing only one label for arrays. The authors have differentiated privacy sensitive data sources (taint sources) into four categories: high bandwidth sensors, low bandwidth sensors, information databases and device identifiers. The high bandwidth sensors include camera and microphone while low bandwidth sensors include accelerometer and GPS (location). The address book and SMS messages are stored in files or databases (termed as informational databases sources) and file level tracking is done for such sources. The information that identifies a user or a device is termed as device identifiers that include the phone number, IMSI, and device IMEI etc. TaintDroid considers the network interface as a taint sink. It looks for transmission of tainted data through the network interface.

TaintDroid has been tested on a set of thirty different applications. These applications were randomly chosen among the 50 most popular applications lists of 12 different categories. The test bed used was Nexus One phones running Android 2.1. During execution of thirty applications, information flow from sensitive data sources was tracked and logged. The experiment lasted for more than 100 minutes and over 22000 network packets were generated in the process. The packet trace on WiFi interface was logged using `tcpdump` that helped in validating the results.

TaintDroid correctly identified 20 different applications that were performing potential privacy violations. These violations included: sending device identifiers, phone information and location information to remote servers. The authors manually labeled the flagged TCP connections and packets. They reported zero false positives (although the possibility of missing false negative exists due to difficulty of identifying them). The authors report a 3% overhead in an application’s load time. The overhead of propagating file taints is significant: address book modification (5.5% to create and 18% to edit). The call set up time increased by 10%. Taking a photo with the phone camera took 29% additional time. Overall, the authors report an average of 14% processing overhead in executing Java instructions and an average of 4.4% memory overhead. Interprocess communication was slowed by 27% with a memory overhead of 3.5%. The major limitation of TaintDroid is that it only looks for explicit information flow (through data); therefore, it is still possible to circumvent taint propagation through implicit flow of information.
2.8.15 AppInspector

Information flow tracking tools such as TaintDroid [Enck et al., 2010] can result in a large number of false positives as the outward flow of information may not necessarily indicate a privacy violation. The authors of TaintDroid [Enck et al., 2010] have recognized this problem and proposed a solution called AppInspector [Gilbert et al., 2011a]. AppInspector is a tool that automates privacy testing and validation for smartphone applications. The proposed framework consists of four components: input generator, execution explorer, information flow tracking and privacy analysis tools. An application is installed in a virtual environment of the system. The automated inputs are supplied using input generator to provide maximum path coverage on the application. The authors propose a mixed of symbolic and concrete execution approach (termed as concolic execution) to provide better code coverage than random testing and execution. The execution explorer keeps track of the execution path; while information flow tracking keeps track of the flow of sensitive data and creates associated logs. The logging of the complete execution path – leading to a leak of sensitive data – might pinpoint the root cause of a privacy violation. The privacy analysis tools classify log activities as normal or violating privacy. The authors have proposed correlation of the logs with the privacy policy (EULA) of applications to automatically identify if flow of sensitive information violates a policy. The authors present the tool as a work-in-progress; therefore, its performance results are not available.

2.8.16 Android Application Sandbox (AASandbox)

AASandbox [Blasing et al., 2010] is a tool for hybrid detection tool for Android smartphones. It detects malware by combining static and dynamic malware analysis techniques. Static detection is done by using malicious pattern searching technique. Afterwards, the application is installed and run on an emulator within a sandboxed environment and heuristics are used to classify the collected features’ set. In static detection, the application package (before it is installed) is decompressed and the binary executable and Android Manifest file are extracted. The manifest file contains important information such as permissions granted to an application at launch time. The executable (containing Dalvik code) is decompiled to get human readable code. Finally, the disassembled code is searched for malicious patterns from previously known malware. AASandbox searches for malicious patterns: malicious code blocks, malicious API calls, a combination of permissions, call to the native runtime environment, attempts to bypass the permissions, or attempts to use services or provisions that can quickly deplete the battery.

AASandbox performs dynamic malware detection through sandboxing within an emulator on a remote server. The Application Sandbox is installed on an Android emulator. It runs in the kernel space and hooks the system calls. When the application makes a system call, AASandbox intercepts the call, executes the original call and logs the call and results. Finally, it returns the results of the original call to the calling application in a user space. The Sandbox has been created as loadable kernel module (LKM) and can be installed on Android device at runtime.

AASandbox has been tested on a dataset of 150 most popular applications downloaded from the Android Market in Oct 2009. The authors created a custom malware
(ForkBomb\textsuperscript{1}) for testing the sandbox. The authors show that the system call logs of ForkBomb are significantly different from those of normal applications. AASandbox has not been tested on a large dataset and it does not use machine learning classifiers. The system can be implemented in a cloud environment that helps in providing decoupled security. The authors have not done any analysis of the time and memory overhead imposed by the detection system.

\subsection{XManDroid: Framework for Mitigation of Privilege Escalation Attacks}

XManDroid [Bugiel et al., 2011] is a security framework for Android that detects and prevents privilege escalation attacks at an application level. The framework detects transitive permission usage of applications at runtime and monitors interprocess communication through standard and covert channels. The detection and prevention of permission leakage is governed through a system-centric security policy.

Android operating system implements a middleware on top of a customized Linux kernel. This middleware manages the installation, configuration, service provision, permission management and interprocess communication for the Android applications. XManDroid framework consists of an application installer, a policy installer and a runtime monitor. It maintains a system-wide view that correlates information from the three components and represents the complete picture of a system in the form of a graph. The application installer updates the system-wide view with the permissions information of new applications. The system policy installers keep the system view updated by adopting the latest security policy. The authors have designed a security policy which consists of rules based on the permission contexts of the communicating applications. The runtime monitor verifies each interprocess communication call against the security policy. A reference monitor implements mandatory access control for interprocess communication between the applications. XManDroid extends the Android’s reference monitor to detect and prevent interprocess communication calls that violate the defined security policy. The reference monitor applies the security policy to direct Intent calls as well as pending and broadcast Intents. A decision maker component helps the runtime monitor in the verification process.

The XManDroid framework also detects and prevents the use of covert channels for permission leaks. It performs this task by monitoring the content providers and the system services that can be used as covert channels. It maintains a track of reads and writes to the content providers and disallows any read from a content provider that had a previous write operation such that the \(<\text{read},\text{write}>\) pair could result in violating the security policy. Similarly, a consecutive \(<\text{write},\text{read}>\) pair also constitutes a violation of the security policy.

The authors tested XManDroid on a Nexus One phone running Android 2.2.1 kernel. A set of 50 benign applications has been collected from Android store. The authors tested their framework on a custom developed set of malicious applications. These applications employ seven different attack scenarios for privilege escalation. The authors have also developed a security policy – consisting of 7 different rules – for these scenarios. All attack scenarios were successfully detected by the framework.

\textsuperscript{1}ForkBomb replicates itself indefinitely to create sub-processes in an infinite loop that results in a denial of service attack.
This is not surprising because the custom malware, developed by the authors themselves, might have resulted in an overfit. Other than automatic testing, the authors have also arranged a manual testing by a group of 20 students. The authors report an average of 3% false positives (incorrectly denied interprocess communication requests). This rate is relatively high for smartphone users and the system policy needs to be optimized to reduce the false positives. The authors report an average latency of 13.13 ms for interprocess communication requests that are not found in the cache and 0.11 ms for the ones found in the cache. The latency appears to be high if we consider the fact that typical runtime for interprocess communication is on the average 0.184 ms. The authors performed a usability test with a group of 20 students and report that users noticed degraded performance of the system but it was nevertheless usable. XManDroid has small memory overhead (maximum memory usage of 4 MB) and it is possible to trade more memory for achieving reduced latency.

2.8.18 Quire

On a smartphone, different applications can communicate with one another and depute tasks through publicly defined interfaces (e.g. Intents on Android). This allows an application to launch a confused deputy attack by improperly calling another application’s interface and forcing it to use its privileges. Moreover, any application (with network permissions) can create an outgoing connection to a remote service; as a result, the id of the source of a network connection can be obfuscated. Quire [Dietz et al., 2011] has been implemented on the Android operating system and it solves two problems: (1) inappropriate use of an application’s permissions through its interface; and (2) trusted communication between applications by using network RPCs (remote procedure calls).

Quire enables an application or a service, receiving an interprocess communication (IPC) message, to be able to see and verify the entire path of an IPC chain. Using this information, the application/service can protect sensitive information from being misused; as a result, an application without privileges cannot trick an application with the sensitive data) to disclose its data. Quire uses cryptographic message authentication codes (MACs) to protect the integrity of the data across IPC and RPC channels. The keys are shared using a trusted OS service (Authority Manager). This enables the operating system to verify the authenticity and integrity of the network RPCs on behalf of the application.

To demonstrate the effectiveness and usefulness of Quire, the authors have developed two applications. The first application uses Quire as a click fraud protection mechanism for an advertising system. Instead of bundling the advertising system as a part of the application code (allowing modification of advertisement library and click frauds), the authors create a child advertisement application and then extend the UI layer to deliver UI events (such as clicks) to the child application. The two applications are stacked such that the transparency in the parent application allows a user to see the advertisement in the child application. The user events are transmitted directly to the child application which can leverage Quire to verify the click event, its source and its freshness (using a time stamp). This allows an advertisement service to ensure that only legitimate clicks result in generating revenues for an application’s publisher.
The second application demonstrates the use of Quire in a micro-payment application (PayBuddy) for verifying the involved parties. A remote service – using Quire’s message authentication code mechanism for IPCs and RPCs, and secure network connection (https) – can verify: (1) the authenticity and integrity of the original application’s order; (2) the fact that the PayBuddy application approved the order (so an explicit approval of a user needs to be processed); and (3) the request originated from a particular device that has been issued with a certificate. This allows distrusting parties (original application, PayBuddy application and the remote payment service) to communicate with each other and validate the sequence of events with the help of Quire; as a result, a user’s consent and the authenticity of the placed order is verified.

The authors evaluated the performance overheads of Quire framework on Nexus One phone with 1 GHz processor and 512 MB RAM. The message signing overhead is $20\mu s$ plus $15\mu s$ per kilobyte. The message verification takes $556\mu s$ plus $96\mu s$ per kilobyte that is significantly high. The overhead of tracking call chain is around $100\mu s$ per hop which is insignificant for small number of hops, but might become significant for large number of hops (hops are usually few). The RPC calls have an overhead below 5 ms even for a chain of 8 distinct applications and this is reasonable. One drawback of Quire is that it doesn’t track two malicious applications that are collaborating to circumvent permission restrictions. The tool is only able to detect that a benign application is being misused by a malicious application.

2.8.19 Stowaway - Android Permissions Demystified

Stowaway is a tool to detect over-privileged compiled applications on Android platform [Felt et al., 2011a]. The authors have extracted the API calls, invoked by an application, and correlated them with the permissions used or acquired by the application. Each application must use a set of least permissions according to its invoked APIs. But applications become over-privileged due to careless unsafe code written by developers. Stowaway builds permission maps to determine content providers and their intents; as a consequence, it is able to determine the privileges that are mandatory for an application to execute successfully.

The DEX executables are decompiled using a known disassembling tool dedexer and are provided to Stowaway tool. The tool extracts permissions from both executables and the manifest. It also tracks and extracts standard API methods. Afterwards, it separates the user defined classes that inherit methods from Android classes. The authors have used different heuristics to handle the problems of Java reflection: (1) the return value of an object is mapped with input parameters; and (2) the type of variable is determined dynamically at the time of its usage. Moreover, the applications which try to access a web address must have Internet privileges. Furthermore, in the Android environment, SD card write permissions are implemented in the kernel; therefore, in Stowaway tool, SD card access is identified by searching sdcard string in an application’s strings and xml files or by checking if the APIs return path to the sdcard. The authors have employed a third party tool – ComDroid [Chin et al., 2011] – to analyze the permissions of sending and receiving intents.

A dataset of 964 Android applications has been used for evaluating the system. Initially, 24 randomly chosen applications are used for training the Stowaway tool.
40 out of remaining 940 applications were analyzed using the tool. It labeled 18 i.e. 45% applications as over privileged with 42 extra permissions. Afterwards, these applications are manually inspected by the authors. It is identified that 17 applications with 39 permissions are over privileged and the tool’s false positive rate stands at 7%. In the remaining 900 applications, a set of 323 applications (35.8%) are marked as over privileged by the tool.

2.8.20 Categorization of Android Applications using static features (CAASA)

This framework uses machine learning algorithms to classify different applications on Android smartphones [Sanz et al., 2012]. The tool extracts the features from Android applications by decompressing them with Android Asset Packaging Tool and using the files contained in .apk file: (1) the existence frequency of printable strings (known methodology of x86 platform for malicious executables detection [Schultz et al., 2001]); and (2) the permissions set owned by the application along with its obtained rating from the Android market. The extraction component uses AndroidManifest.xml to extract permissions and features of the device invoked by the application. Afterwards, the dedexer tool is used to disassemble the executable. As a result, a directory structure is produced with recognized classes. Finally, the strings are extracted from the application. The term frequency and inverse document frequency are calculated for the extracted strings. The authors have obtained the market information by using an open source API android-market-api that provides features: the number of ratings of an application, the number of times an application is installed, and the user ratings of the application.

In the classification phase, the authors have employed four well known machine learning classifiers: bayesian networks, C4.5 decision tree, k-nearest neighbor and support vector machine. A ten-fold cross validation is used for training and testing. The proposed scheme is empirically evaluated using 820 Android applications belonging to 7 distinct categories. The authors report the accuracy of the system by using area under the ROC curve (AUC) measure. The Bayesian network classifier provides the best accuracy with an AUC of 0.93. The authors have claimed in the paper that the proposed scheme is suitable for malware detection but they haven’t performed any experiments on malware samples. The authors have not reported processing and memory overheads.

To conclude, in this section, we have analyzed the existing tools for detecting malicious applications. In Table 2.2, we present a summary of malicious application detection techniques for a quick reference.

2.9 Misc. security & privacy solutions for smartphones

The authors of [Lange et al., 2011] have proposed a secure operating system framework that wraps primary smartphone OS in a virtual machine to establish a seclusive environment. The renowned micro-kernel L4 is used for this purpose. The framework runs applications – demanding high security – in parallel with JVM.
Table 2.2: Correlation Matrix of Tools and Features

<table>
<thead>
<tr>
<th>Mobile Platform</th>
<th>Android</th>
<th>CAASA (2.8.20)</th>
<th>Woodpecker (2.8.1)</th>
<th>SFCMD (2.8.2)</th>
<th>PiOS (2.8.4)</th>
<th>Centroid (2.8.3)</th>
<th>Virus Meter (2.8.7)</th>
<th>EGA Anomalies (2.8.8)</th>
<th>Paranoid (2.8.9)</th>
<th>Crowdroid (2.8.10)</th>
<th>KBEA (2.8.11)</th>
<th>Andromaly (2.8.12)</th>
<th>Misisse Detector (2.8.14)</th>
<th>TaintDroid (2.8.14)</th>
<th>AppInspector (2.8.15)</th>
<th>AASandbox (2.8.16)</th>
<th>XManDroid (2.8.17)</th>
<th>Quire (2.8.18)</th>
<th>MADAM (2.8.6)</th>
<th>SmarDroid (2.8.5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>iOS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Symbian</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Windows Mobile</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Environment</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>On-Host (2.4.1)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Decoupled (2.4.1)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Kernel Mode (2.4.2)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>User Mode (2.4.2)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Sandboxing (2.4.3)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Emulation (2.4.3)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Analysis Type (2.4.4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static Analysis</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Dynamic Analysis</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Hybrid Analysis</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Classification</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Anomaly Based (2.4.5)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Two Class Classification (2.4.5)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Signature verification</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Control flow Graphs</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Features for Analysis</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static Function Calls</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>System Calls</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>File Features</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Info Flow Track</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Text Strings</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Applications Permission</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>OS &amp; File Sys Events</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Instruction Traces</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Concolic Execution</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Power Utilization</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>IPC monitoring &amp; authentication</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
In [Zhou et al., 2012], a tool – DroidMoss – is developed to detect packed applications that are launched at six unofficial android application markets. Some of these applications (5 to 13%) are simply repacked versions of the applications that are already available at the official Android market. The authors concluded that repacking is used to replace existing advertisements with new ones to earn revenues in an illegitimate fashion. DroidMoss employs a fuzzy hashing technique to compute similarity score of different applications and use it to flag packed applications.

RGBDroid [Park et al., 2012] is a response based tool that can detect privilege escalation attempts by identifying processes that try to obtain illegal root privileges and restricts access to protected resources. The tool restricts illegal activities after a security breach that results in reducing the processing overheads which are typically caused by protective approaches.

In [Yan and Yin, 2012], the authors have proposed DroidScope which is a multi-level semantic analysis tool that performs dynamic profiling and information tracking to detect malicious behavior and privacy leaks in Android based smartphone applications. The tool runs in a virtualized environment and logs instruction traces, API calls (at OS level and Dalvik VM level) and uses taint analysis to discover leak of sensitive information. The tool has been tested on two real world malware samples.

A security solution I-ARM-Droid is proposed in [Davis et al., 2012] for protecting smartphones from malicious or non trusted applications. The framework monitors the APIs and associated security polices to identify the violations. Later, the Dalvik byte code is rewritten that interpolates the existing methods to enforce the least required permissions or security policies. In case a method in class C is to be interposed, the framework generates a class W (wedge class) that extends the class C. The new W class contains the stub and wedge methods that correspond to the target methods. The extended classes from class C are identified and modified to extend the W class. The authors have demonstrated the compatibility of their rewritten code with the Android platform. They have reported 110 ms average processing overhead in rewritten applications while the size is increased by 2%.

Jana [Lee et al., 2012] platform sandbox aims at addressing a users’ privacy concerns. Its distributed architecture safely transfers a user’s sensitive information with dependable privacy protection mechanisms; as a result, the responsibility of providing privacy is shifted from applications to the framework that creates a sandbox for every application run by each user – spanning from smartphones to the systems on a cloud. The sandbox provides dedicated communication and storage channels, with customized privacy preserving methods, to enable smartphones applications to do useful tasks. The authors have developed a prototype to demonstrate different features of Jana.

In [Bugiel et al., 2012], a security framework for Android OS is presented that takes care of confused deputy processes – the vulnerable interfaces of some privileged or over-privileged application processes that are misused by malicious processes to perform their covert activity – and collusive attacks: malicious processes embed them into benign applications to misuse them to perform malicious activities not allowed by their personal permissions. The authors have built a cross-layer, system-oriented and policy-based architecture that scrutinizes the communication channels in realtime among applications. At an intermediate layer, direct and indirect (using OS components) inter-process communication (IPC) is controlled and QUIRE-like
links are setup between communicating processes to validate the call chain by using reference monitors. Moreover, the authors impose access control procedures on the file system and all types of sockets in the kernel. A feedback channel – setup between the kernel and an intermediate functional layer – implements the policy at a lower level. Finally, the effectiveness of the framework is determined with the help of an application that launches realtime attacks.

A static analysis solution to detect over-privileged applications (on Android platform) with the help of the permission gap – difference between granted and needed permissions – is presented in [Bartel et al., 2012]. The proposed tool is tested on two datasets: it detects 13% of 742 (dataset 1) and 5% of 679 (dataset 2) Android applications that suffer from a privileges gap. A significant amount of research is focused on detecting over-privileged applications on smartphones (an interested reader can refer to [Gilbert et al., 2011b], [Grace et al., 2012b], [Gibler et al., 2012] and [Wetherall et al., 2011]).

An interesting article [Felt et al., 2012] provides a survey regarding smartphone users’ behavior and their awareness about the role of permission settings in Android during the process of installing applications. Unfortunately, the Android’s permission system significantly depends on the risk-awareness of its users: they can install an application if they agree with the demanded permissions at the install time; otherwise, they have an option to abort the installation. The authors conducted a survey of 308 online users and 25 users in their laboratory asking them about their sensitivity, knowledge and conduct towards permission settings during the process of installing applications. It is amazing to see that only 17% users paid attention to the permission during installation and a meagre 3% could only correctly answer three basic questions involving comprehension of permissions. The conclusion is: the Android’s permission mechanism is inappropriate for a vast majority of Android users because it fails in enforcing correct privileges. Consequently, they suggested improvements for a better usability experience.

A runtime verification system for Android is proposed in [Bauer et al., 2012]. It maintains a profile of suspicious applications and if the profiles of running applications match with the suspicious one, an alarm is raised. Take the example of an application that gets executed at the boot time, it requests the location from GPS, and finally, it connects to the Internet to send the mobile’s location to a remote host. In this case, the framework automatically activates a monitor to generate an alert, if the sequence of such events is followed by any user application.

An OS service MoRePriv [Davidson and Livshits, 2012] provides omnipresent personalization like the location service support and should be implemented in the kernel space instead of a user space. The service parses smartphone users’ information streams over the Internet – a users’ email, SMS, social networking database and usual network communication information flows etc. – and builds a user’s profile to preserve his privacy by providing filter hooks to protect information leaks. MoRePriv empowers a user to organize his preferences in different user applications by exposing relevant personalization APIs. Moreover, the service also enforces privacy constraints for advertisements in the applications on the basis of a user’s preferences profile. The authors have conducted experiments to demonstrate that MoRePriv helps in reducing over-permissions in 73% of tested user applications.

Advertisements are (mostly) an integral part of smartphone applications to generate revenue. To embed advertisement services, binary libraries are shipped with
the smartphone applications. The binary libraries require execution permissions and demand host applications to share their sensitive information with them. In [Pearce et al., 2012] and [Nunez, 2011], the authors have reported that 49% Android applications contain one binary library (on the average) for advertising services and 46% applications are able to subscribe over-permissions because of these libraries. They also show that 56% applications (having embedded libraries) also request the location information without the consent of a user. The authors have developed AdDroid that filters and isolates advertisement libraries and their requested permissions; as a result, they can show advertisements without tricking a user to enable sensitive permissions or share privacy related information. The other approaches that also isolate advertisement libraries (and their requests for sensitive information) from normal user applications are presented in AdSplit [Shekhar et al., 2012], [Leontiadis et al., 2012] and [Grace et al., 2012a].

2.9.1 Other Surveys on Smartphone Security

A survey paper on security of mobile devices is published by [La Polla et al., 2012]. In the paper, the authors have briefly discussed mobile wireless technologies: GSM, GPRS and EDGE along with networking technologies WLAN and bluetooth etc. The attack vectors – wireless interface, buffer over flow, network infrastructure, virus and worms, user behavior, privacy, denial of service, battery depletion and over billing attacks etc. – on mobile devices have been discussed. They have classified attack detection methodologies into five different categories on the basis of detection principles, architecture of security solutions, threat detection modes, data collection strategies and mobile operating systems. These models are briefly summarized here: (1) detection principles used anomaly based algorithms, machine learning classifiers, algorithms modeling energy consumption, conventional signature based models etc; (2) the architecture defines the point of deployment of the technique: host-based or distributed solution in the cloud; (3) the mode determines the countermeasures will be active or passive; (3) detecting intrusions by training the framework on different types of datasets: system calls dataset, system performance counter and keystrokes etc.; and (4) finally, the security solution is developed for a particular mobile operating system – Android, iOS, Symbian etc.

In [Enck, 2011], the author has presented different detection techniques for smartphones. The techniques are classified on the basis of their protection mechanism and application analysis. The techniques for system and applications’ policies, platform security (e.g. visualization etc.), multiple user access and information faking are categorized as “protection mechanisms”. The dynamic, static, permissions and cloud based analysis approaches have been summarized as “application analysis”.

In another survey [Elfattah et al., 2012], the authors have categorized mobile malware detection techniques into three topologies: (1) device based detection; (2) infrastructure based detection; and (3) hybrid topologies. In the first category, they discussed behavioral and access control models on mobile hosts. In the second category, a mobile host tracks and logs the communication messages (or packets) and transfers them to a remote proxy server for detecting malware on a single or multiple mobile devices concurrently. In the hybrid topology, malware detection is done in a distributed manner by placing the security solution on host and infrastructure; as a result, the processing overheads on a smartphone are minimized.
The authors of [Becher et al., 2011] have provided a brief overview of security challenges on different smartphone interfaces (mobile network security problems: attack vectors and vulnerabilities using browsers and backup mirror servers, DOS attacks etc.). Some vulnerabilities are hardware dependent (like smart card) and some are generic and platform independent: link encryption, connectivity and handshakes, SMS and MMS vulnerabilities etc. The examples of software-based vulnerabilities are: malicious software, identity theft, browsers problems, mobile botnets, SMS and MMS vulnerabilities etc. In another survey of mobile malware [Felt et al., 2011b], a behavioral analysis of available mobile malware and existing protection mechanisms and emerging trends are discussed. They have focused their survey on root exploits for Android. They have also described different exploits and how they could be used in combination with root exploits.

In survey of [Yan et al., 2009], the authors have summarized industrial and academic research about different paradigms for detecting mobile malware: (1) monitoring power consumption patterns for benign and malicious applications; (2) using a dual mode approach (different addresses for APIs in development and execution stages) for smartphone applications is proposed. Moreover, they have proposed the idea of hardware sand-boxing (whenever a user is busy making a voice call, the hardware modules needed for Internet access must be disabled.)

The survey [Vinod et al., 2009] briefly summarizes different paradigms for smartphones. But, two methodologies have been discussed at depth: (1) signature-based malware detection; and (2) detection by de-obfuscating the obfuscated code. The other relevant surveys are [Damopoulos et al., 2012] and [Schmidt and Albayrak, 2008].

Most of the published surveys have not considered the subject of security on smartphones in depth and breadth demanded by ever increasing complexity and sophistication of smartphone vulnerabilities. (Most of the surveys focus only on a specialized area – infrastructure base security – and ignore highly relevant security and privacy threats. In this survey, we have attempted to cover the complete spectrum of challenges in mobile security: mobile malware and their types, infection vector and vulnerabilities, challenges for smartphone security systems, comprehensive analysis of malware and privacy leaks detection, security tools and their features relevant to the OS platform, detection accuracy and the false alarm rate of an approach, and overheads in terms of processing.

The major contributions of this survey chapter are: a focus on the niche of malicious applications detection related to security and privacy on smartphone platforms, generalizing a comprehensive malicious smartphone applications detection framework as a guideline for future development by vendors and security researchers, a comprehensive analysis of recently proposed techniques utilized by smartphone security and privacy analysts, and a survey of recently proposed tools using these techniques, along with their significance and shortcomings.

### 2.10 Conclusion

Smartphones are becoming the core delivery platform of ubiquitous “connected customer services” paradigm; as a consequence, they are attractive targets of intruders (or imposters). Researchers have realized that classical signature-based anti-malware techniques are not capable of providing efficient and effective detection tools
against novel, zero-day and polymorphic malware for resource constrained smartphones; therefore, in last couple of years unconventional (non-signature) intelligent solutions, based on behavioral analysis (static or dynamic) have been proposed. In this survey, we have enumerated various types of malicious applications and the infection vectors that are a threat to security and privacy on smartphones. Creating an application for malicious applications detection requires some important challenges to be met and making crucial implementation decisions and we have enumerated these challenges and decisions. A generic framework has been presented for detection of malicious applications on smartphones that helps a reader understand the system wide architecture of malware analysis and detection techniques. We have also presented, analyzed and categorized latest published techniques utilizing static and dynamic detection techniques on smartphones. Finally, we have reviewed the recent malware detection tools and frameworks that utilize these techniques. The literature review and survey presented in this chapter enables us to understand the existing state-of-the-art malware detection techniques – and using their merits and demerits to propose a comprehensive zero-day malware detection framework on smartphones.
Chapter 3

ELF-Miner: Using Structural Knowledge and Data Mining Methods To Detect New Malicious Executables

3.1 Introduction

Linux malware can pose a significant threat – its (Linux) penetration is exponentially increasing especially on smartphone platforms – because little is known or understood about Linux OS vulnerabilities. Now a days, smartphone platforms are providing support for even native code execution [Superphone, 2013]. Therefore, we believe that now is the right time to devise non-signature based zero-day (previously unknown) malware detection strategies before Linux intruders take us by surprise. Therefore, in this chapter, we present a malware detection framework based upon the static analysis of structural headers of Linux executables. This framework is portable (after cross-compilation) to the smartphones which provide native code execution support.

Linux – due to its open source nature – is getting an ever increasing attention both by researchers and developers [Linux-Share, 2010]. Moreover, home users and business enterprises are preferring Linux based Personal Computers (PCs), server machines and smartphones etc. As a consequence, Linux will definitely become a favorite target for hackers, the moment its market share makes it an attractive proposition to launch attacks on Linux running hosts. The current scarce availability of Linux malware has also lead Linux security experts to hold a notion that Linux is inherently secure [Fedora, 2010]; therefore, malware detection on Linux has never received its due attention. Consequently, Linux based computers – open source nature makes the task of a hacker also easier – are not adequately protected against emerging threats.

In this chapter, we first do a forensic analysis – with a particular focus on the structural information present in the header of an ELF file – of available Linux malware. The analysis helped us in identifying a set of structural features that can be used to discriminate a malware file from a benign one. We then apply preprocessor filters to remove redundant features. Finally, we give the remaining features’ set as
3. ELF-Miner: Using Structural Knowledge and Data Mining Methods To Detect New Malicious Executables

an input to a number of classifiers – classical machine learning and bio-inspired – to finally detect malware. We compare the accuracy of different classifiers on a collection of 709 malware samples available from *vx heavens* [VX-Heavens, 2009] and *offensive computing* [Offensive-Computing, 2009]. Our results show that our system is able to detect Linux malware with more than 99% accuracy. The true strength of our approach is that it does not make a signature from the instructions of a malware; therefore, it is a non-signature based technique and has the ability to detect zero-day (on the day of its launch) Linux malware. Our data mining approach takes only a fraction of a second; therefore, we can deploy it in realtime on Linux systems

The rest of the chapter is organized as follows. In the next section, we briefly summarize related work. Section 3.3 discusses the ELF mined features’ set in detail. Section 3.4 provides an overview of ELF-Miner framework. A brief introduction of malware dataset is given in Section 3.5. A comprehensive forensic analysis of benign and malware files is provided in Section 3.6. An overview of our classification methodology has been presented in Section 3.7. Section 3.8 presents the experiments, results and evaluation of classifiers on the dataset. Finally, Section 3.9 presents the conclusion of the chapter and future research directions.

### 3.2 Related Work

In the related work section, we summarize relevant non-signature based malware detection techniques based on static analysis of executables. All of these techniques are for windows executables. Recently Zubair et al. proposed a technique PE-Miner [Shafiq et al., 2009a] based on static analysis that extracts structural information from the headers of a windows executable and uses it to detect malware. Later, they enhanced it in [Shafiq et al., 2009b] to overcome the side effects of doing packing (encryption) of executables. As a result, the enhanced version has the capability to discriminate packed malware files from packed benign files and unpacked malware files from unpacked benign files. In this chapter, our focus is to explore another dimension of research: whether using the structural information to detect malware could be generalized to executables of other operating systems? This chapter proves this thesis.

The other recently proposed malware detection techniques for windows executables are: Perdisci et al. [Perdisci et al., 2008], Schultz et al. [Schultz et al., 2001], Masud et al. [Masud et al., 2008] and Kolter et al. [Kolter and Maloof, 2004]. We now briefly summarize their detection methodology but an interested reader can refer to [Shafiq et al., 2010] for a detailed description.

The technique proposed by Masud et al. [Masud et al., 2008] utilizes a hybrid features selection scheme to detect malicious PE files. The hybrid set consists of three types of features: (1) n-grams as binary features, (2) n-grams of unique assembly instructions, and (3) n-grams of dynamic link library calls. Decision trees are used to classify the PE files on the basis of the extracted hybrid features’ set. The technique uses a disassembler to create the features’ set; therefore, the scanning time is significantly large (making it difficult to use it in realtime).

---

1Initial work on this approach has been published as “PE-Miner” [Shafiq et al., 2009a] that detects zero-day malicious executables on Windows platform using the structural information in the header of a PE file.
3.3 The Features’ set of ELF-Miner

In [Perdisci et al., 2008], Perdisci et al., the authors proposed a framework ‘McBoost’ based on two level classification. The two classifiers – C1 and C2 – have been used to classify the non-packed and packed portable executables (PE) files. They developed a customized unpacker to extract hidden codes from an executable and provide them to the C2 classifier for analysis and classification. In [Schultz et al., 2001], Schultz et al. proposed three different methodologies for detecting malicious executable files on the Windows platform. The first technique is based on the information of dynamic link libraries (DLLs), their function calls and citation counts. In the second technique, they use the strings as binary features (i.e. present or absent). The third technique uses two byte words (instead of a string) as binary features. Later on, Kolter et al. [Kolter and Maloof, 2004] have improved the third technique of Schultz et al. by using 4-grams as binary features. The authors of PE-Miner have already compared the techniques of Schultz et al., Perdisci et al., and Kolter et al. with PE-Miner in [Shafiq et al., 2009a] – in terms of accuracy, false alarm rate and scanning time (detection delay). The end results of their experiments show that none of these techniques are realtime deployable because they have: (1) less than 99% detection accuracy, (2) more than 1% false alarm rate, and (3) large file scanning time (order of seconds). In comparison, they establish the supremacy of PE-Miner because it achieves greater than 99% detection accuracy with less than 0.1% false alarm rate and its scanning time is comparable (200 milliseconds) with that of commercial antivirus software as well. To make the chapter self contained and to verify the comparative results of ELF-Miner with other static analysis based techniques, we compare it – in terms of classification accuracy (AUC) only – with the techniques: (1) McBoost, and (2) Kolter’s technique (KT) (4-grams as binary features).

3.3 The Features’ set of ELF-Miner

In this section, we present an overview of structural features – extracted from the header – of benign and malware ELF files. In order to make the chapter self contained, we briefly introduce ELF format.

3.3.1 Executable and linkable format (ELF)

In Linux, the object files play a key role in the process of program linking and execution. The manual of ELF format [TI-Standards, 1993] describes three types of ELF files: (1) relocatable file (it contains the “how to link” information with other object files in order to create a shared library or an executable file), (2) executable file (it contains data and information required by an operating system to create a program image that can be executed by accessing information in the file), and (3) shared object file (it contains all the information required for both static and dynamic linking). We have used both shared and executable ELF files in our dataset.

The structure of an ELF file is as shown in Figure 3.1. An ELF header, at the beginning of every file, holds a blueprint of a file’s organization. In case of a relocatable file, a section header table is mandatory and a program header table is optional. In case of an in-executable object file, a program header table is compulsory and a section header table is optional. Note that a section header table contains the description of all sections that exist in the object file. Every section in the object
3. ELF-Miner: Using Structural Knowledge and Data Mining Methods To Detect New Malicious Executables

Figure 3.1: Executable and linkable format (ELF) structural view [TI-Standards, 1993]

file has an entry in this table. The section entries provide the attributes of a section: section name, section size etc. The sections primarily hold the object file information required for building and linking ELF programs. The information about program instructions, data, symbols, dynamic linking and relocation is contained in different sections of an object file. We skip further details of ELF format for brevity but an interested reader can find them in [TI-Standards, 1993].

3.3.2 ELF Structure-based Features’ set

We initially extract 383 features from the header of an ELF file (see Table 3.1). Later we use our forensic analysis to rank these features to select the ones having the best potential to discriminate a benign file from a malware file. We now introduce our features’ set in detail.

**ELF Header.** ELF header consists of data structures that describe the organization of an ELF file. The header structures help in parsing an ELF file. An ELF header structure consists of a number of typical fields: identification, machine type, ELF file version, entry point address, program header table file offset in bytes, section header table file offset in bytes, processor specific field flags, ELF header size in bytes, size and count of individual entries in program header table, section header size in bytes, number of entries and index of entry associated with the section name string table in section header table. We use 16 fields of ELF header excluding program and section headers offsets in our features’ set.

**Section Header.** The structure of a section header contains the fields: section name, section type, section flags field that represents miscellaneous attributes, address field where section’s first byte should reside, section’s offset field that holds the

<table>
<thead>
<tr>
<th>No</th>
<th>ELF Structure Name</th>
<th>Features Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ELF Header</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>Section Headers</td>
<td>238</td>
</tr>
<tr>
<td>3</td>
<td>Program Headers</td>
<td>40</td>
</tr>
<tr>
<td>4</td>
<td>Symbols Section</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>Dynamic Section</td>
<td>29</td>
</tr>
<tr>
<td>6</td>
<td>Dynamic Symbol Section</td>
<td>17</td>
</tr>
<tr>
<td>7</td>
<td>Relocation Sections</td>
<td>26</td>
</tr>
<tr>
<td>8</td>
<td>Global offset table</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>Hash table</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>Total Fields</td>
<td>383</td>
</tr>
</tbody>
</table>

Table 3.1: Features extracted from ELF files
first byte of section’s offset from the beginning of ELF file, size of section in bytes, *section header table* index link, section info field that depends upon section type, section address alignment field that shows the alignment constraints. We include 238 fields of 34 section header structures in our features’ set, excluding the section names, addresses and offsets fields.

**Program Header.** All executable and linkable files are array of structures, which represent program segments and other information that are mandatory for successfully executing the program. Each executable file segment may contain one or more sections. The segment header structure contains 8 fields: segment type, segment offset that gives the offset from the beginning of the file, virtual address at which first byte resides in the memory, segment’s physical address, file size field provides the number of bytes in the segment file image, memory size field contains the number of bytes in memory image, flags’ field holds the flags related information of segments, p_align member contains alignment information of segment in file and memory. We select 40 features excluding all addresses and offsets fields of first 8 segments for our features’ set.

**Symbol Table.** It contains the information related to symbols in an ELF file but this information is not mandatory in all types of object files. The symbol table structure has five major fields – name, value, size, info, other and section header index – but the number of entries vary in symbol table of different executable files. Therefore, we make categories on the basis of st_info field – it contains the information of symbol’s binding and attributes. The resulting categories are: total number of [symbols, local symbols, global symbols, weak symbols and stb_lo_proc symbols]. The objects and functions symbols are further categorized on the basis of their scope i.e. total number of [local objects, global objects, weak objects, local functions, global functions, weak functions, sections, files, stt_lo_proc and stt_hi_proc] objects. As a result, we create 17 categorical fields from the symbol table for our features’ set.

**Dynamic Section.** If any ELF file has dynamic linking information, segment PT_DYNAMIC (dynamic section is an integral part of it) becomes part of the program’s header table. The structure of dynamic section contains a 4 bytes field d_tag and union d_un. All fields are classified on the basis of the information available in parameter d_tag. Like symbol table, dynamic section also doesn’t contain a fixed number of entries; as a result, all these entries are classified into 27 categories for our features’ set.

**Dynamic Symbols Section (DSS).** If an object file contains dynamically linked objects, it may have dynamic symbol section also. The details of dynamic symbol section are similar to that of the symbol section as mentioned earlier in this section. 17 features are extracted from the DSS section.

**Relocation Section.** In ELF, the process of relocation links referred symbols with their definitions. When a program calls a function, the control must be transferred to the function definition at an appropriate address. The structure of .rel contains two members – offset and info. The offset field refers the location where the relocation action should be performed. In case of relocatable files, the offset field contains the offset from the beginning of section to the storage unit of relocation. If the files are executable or shared objects, an offset represents the virtual address of the storage unit. The info field maintains information about the type of relocation and the symbol table index that is used for relocation. We extract 26 features from the contents of sections .rel and .rela for discriminating the benign and malicious
Global Offset Table (GOT). This table contains absolute addresses of objects representing private data. It provides address without affecting position independence and sharing capability of a program text. The programs refer to GOT for absolute address values using position independent addressing. For our features’ set, we only have one field that represents the size of GOT.

Hash Table. It is a 32-bit object table that facilitates the access to the symbol table. Only one field of the hash table size is included in our features’ set.

3.4 ELF-Miner Framework

We now discuss the architecture of our proposed framework – ELF-Miner – that mines the structural information in ELF executables to detect malicious ones. Our framework consists of three basic components: (1) features extraction, (2) features preprocessing, and (3) classification (see Figure 3.2). The feature extraction block first does a validity check to confirm the validity of an ELF header. If the file is legitimate ELF executable, it extracts 383 features from its header. Afterwards, the preprocessing block ranks these features to filter redundant features or features with smaller classification potential; as a result, the remaining features’ set is given to a number of classifiers that eventually classify the executable as malicious or benign.

Figure 3.2: Block diagram of ELF-Miner framework

3.5 ELF Malware Dataset

In this section, we present an overview of the datasets used in our experiments. In order to remove any bias because of the size of files, we divide ELF files – both malicious and benign – into six bins of different sizes starting from 20 KB to 4 MB. In order to have a balanced dataset, we ensure that the percentage of benign and malicious files in a certain category is approximately the same to remove any bias on the basis of size (see Table 3.2). The benign ELF files are collected from Linux operating system’s directories /bin, /sbin, and /usr/bin. We use the ‘size based filtering’ criteria to select approximately an equal percentage of benign ELF files. As
a result, only those files are selected which have size greater than a given threshold that is calculated on the basis of number of files and their size in a specific malware category. In comparison, Linux malware dataset is collected from vx heavens [VX-Heavens, 2009] and offensive computing [Offensive-Computing, 2009]. We have combined some malware categories because of their similar functionality. For example, we have combined Exploits, Rootkits and Hacktools to create a single Expts + RK + HT category. As a result of the unification process, the number of malware samples per category are increased significantly. We now provide a brief description of individual and combined malware categories – 8 to be precise – to make the chapter self-contained.

**Backdoor + Sniffer (Bkdrs).** A backdoor is a program which allows bypassing of standard authentication methods of an operating system. As a result, remote access to computer systems is possible without explicit consent of the users. Information logging and sniffing activities are possible using the remote access.

**Constructor + Virtool (Cnstr).** This category of malware mostly includes toolkits for automatically creating new malware by varying a given set of input parameters. Virtool and constructor categories are combined because of their similar functionality.

**DoS + Nuker (DoS).** Both DoS and nuker based malware allow an attacker to launch malicious activities at a victim’s computer that can possibly result in a denial of service attack. These activities can result in slowing down, restarting, crashing or shutting down of a computer system.

**Email- + IM- + SMS Flooder (Fldrs).** The malware in this category initiate unwanted information floods such as email, instant messaging and SMS floods. Similarly, well known network packets based attacks – TCP SYN and ICMP floods – are also launched by this category of malware.

**Exploit + Hacktool + Rootkits (Expts + RK + HT).** The malware in this category exploit vulnerabilities in a system’s implementation which most commonly results in buffer overflows. These attacks are launched to take administrative permissions remotely or to execute malicious code on systems.

**Email- + M- + IRC- + Net Worm (Wrms).** The malware in this category spread through instant messaging networks, IRC networks and port scanning.

**Trojans (Trjns).** A trojan is a broad term that refers to stand alone programs which appear to perform a legitimate function but covertly do possibly harmful activities such as providing remote access, data destruction and corruption.

**Virus + Spoofer (Vrs + Spfr).** A virus is a program that can replicate and attach itself with other benign programs. It is probably the most well known type of malware. Similarly, a spoofer is a program that successfully masquerades as another host or program and gains an illegitimate access to the system.

We have recently obtained latest malware dataset from vx heavens [VX-Heavens, 2009] and offensive computing [Offensive-Computing, 2009] which contains more than 900 Linux malware; however, only 709 passed the validity test. The distribution of malware into different categories and the size is shown in Table 3.2. The top four malware categories in a descending order are Bkdrs, Expts, Vrs and RK respectively. Another important observation is that 63% of malware is less than 20KB which signifies the importance of removing the size bias (as a feature) during classification; therefore, we have ensured that 61% of benign files are also less than 20KB in our dataset.
3. ELF-Miner: Using Structural Knowledge and Data Mining Methods To Detect New Malicious Executables

Table 3.2: Benign and malware file size normalization

<table>
<thead>
<tr>
<th>File Sizes</th>
<th>Benign Files</th>
<th>Malware Files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ELFHEENTRY</td>
<td>ELFHEENTRY</td>
</tr>
<tr>
<td></td>
<td>ELFHEPHNUM</td>
<td>ELFHEPHNUM</td>
</tr>
<tr>
<td></td>
<td>ELFHESHNUM</td>
<td>ELFHESHNUM</td>
</tr>
<tr>
<td></td>
<td>ELFHESHSTRIDX</td>
<td>ELFHESHSTRIDX</td>
</tr>
<tr>
<td>20K</td>
<td>445</td>
<td>63</td>
</tr>
<tr>
<td>20-50K</td>
<td>161</td>
<td>25</td>
</tr>
<tr>
<td>50K-100K</td>
<td>97</td>
<td>12</td>
</tr>
<tr>
<td>100K-500K</td>
<td>18</td>
<td>3</td>
</tr>
<tr>
<td>500K-1MB</td>
<td>57</td>
<td>9</td>
</tr>
<tr>
<td>Total</td>
<td>534</td>
<td>107</td>
</tr>
</tbody>
</table>

3.6 Forensic Analysis of Benign and Malware ELF Executables

We now provide our forensic analysis – by closely studying the pattern of the fields in the headers of different ELF files – to understand the difference in the headers of benign and malware ELF executables. The aim is to establish the thesis that the structural information of malware files is different from that of the benign ones. We utilize well known information-theoretic measures – Resistor-Average (RA) divergence [Johnson and Sinanovic, 2001a] and Frequency Histogram analysis – to analyze the difference between various fields in ELF headers of benign and malware files. (Note that we do not rank the identified features, by using information-theoretic measures in our forensic analysis, for classification purposes.)

The definition of RA divergence is given as:

$$ RA(p, q) = 1/\left(1/\frac{KL(p||q)}{KL(q||p)}\right) $$

In equation (3.1), KL (Kullback-Leibler) distance [Cover and Thomas, 2006] is another information-theoretic measure, which can be used to measure the difference between two probability distributions – $p(x)$ and $q(x)$ in equation (3.2). KL distance is always non-negative and is zero only in case of $p=q$. Moreover, in a special case when $p(x) = 0$ or $q(x) = 0$, the distance becomes zero or infinity; therefore, we simply add $\epsilon ps = 2^{-52}$ – which is the distance from 1.0 to the next largest double precision number – in both the distributions $p(x)$ and $q(x)$ to avoid this problem. Moreover, KL distance is not symmetric over two distributions. As a result, we use a symmetric distance measure.

$$ (KL(p||q)) = \sum_{x} p(x) \cdot \log \frac{p(x)}{q(x)} $$

Similarly, the frequency histogram provides the count of occurrence – of field values or headers in benign and malware files separately. In order to abbreviate the large names of fields in an ELF header, we use a convention: the ELF headers’ or sections’ names are used as a prefix with the field name. We will briefly describe the functionality of fields, but an interested reader is recommended to consult [TI-Standards, 1993] for details.

**ELF Header Examination.** The RA divergence values for 16 fields of ELF header for benign and malware are shown in Figure 3.3. It is interesting to note that 4 header fields are significantly different in benign and malware files. The fields are: ELFHEENTRY, ELFHEPHNUM, ELFHESHNUM and ELFHESHSTRIDX. Most of these fields contain distinct numeric values – indices and header sizes – which are significantly different in benign and malware executables. Most of other fields contain either zero or constant values; as a result, RA divergence is zero or negligible.
3.6 Forensic Analysis of Benign and Malware ELF Executables

**ELF Section Headers.** In object files, various sections contain the control and program information. We initially extract headers of 34 sections and their description is provided in Table 3.9 of appendix A and in Section 3.3.

**Sections’ headers frequency histogram analysis.** We have plotted the frequency histogram of different sections – occurring in benign and malware files – in Figure 3.4. It is interesting to note that 13 out of 34 sections never appear in any benign or malware file. These sections generally contain debug information, initialized values, read-only data, and rarely used pre-existing extensions. Some sections – `.comment`, `.note`, `.strtab`, `.symtab`, and `.sbss` – are present in malware files but are generally absent in benign files. Similarly other sections – `.rel.dyn` and `.got.plt` – are not used by most malware programs. We have zoomed at the distribution of these 7 sections in Figure 3.5 to get a better understanding.

Our analysis shows that most malware misuse `.comment`, `.notes` sections (used to hold version control information and file notes respectively) to store their malicious information. It is interesting to see that sections `.symtab` and `.strtab`, which hold
3. ELF-Miner: Using Structural Knowledge and Data Mining Methods To Detect New Malicious Executables

![Figure 3.5: Overall frequency of 7 sections in all malware (left) and benign (right) files](image)

the symbols and strings related information of the programs, are missing in most benign files. We investigated this unexpected trend and found that most of these binaries are stripped and these sections are removed from them [Linux-Commands, 2010].

Similarly, `.rel.dyn` holds dynamic relocation information and is used by the majority of the benign files and almost 35% of the malware files. Section `.got.plt` that holds read-only portion of the global offset table is used by almost all benign files and few malware files. The frequency of all other sections can be observed from the Figure 3.4.

![Figure 3.6: Symbol Table RA divergence graph for benign and malware files](image)

Symbol Table. Similarly, the RA divergence in different fields of symbol table is shown in Figure 3.6. We see that except processor specific symbols – `SYMTABHIPROC`, `SYMTABLOPROC`, `SYMSTHTIPROC` and `SYMSTHTLOPROC` which are not used by benign and malware programs – a significant difference exists between symbol table entries. It is already mentioned that most of benign programs – mostly stripped binaries – do not contain a symbol table. In comparison, most malware have symbol table entries.

Dynamic Section. The graph of RA divergence for dynamic section fields is shown in Figure 3.7. We can see that some fields – with high RA divergence – have the potential to discriminate between benign and malware files. Most of these
3.6 Forensic Analysis of Benign and Malware ELF Executables

Figure 3.7: Dynamic section RA divergence graph for benign and malware files
discriminating fields represent different categories of dynamic elements that are used by benign and malware programs in a different manner. For example, the first field – DYNTOTAL – with a RA value of 0.15 shows that total number of dynamic symbols in both types of files are significantly different.

Figure 3.8: Dynamic symbol section RA divergence for benign and malware files

Dynamic Symbol Section. The RA divergence of different fields of this section is plotted in Figure 3.8. The figure shows that other than the processor specific features only three categories of attributes – STTFILE, STOBJSTBLOCAL and STTFUNSTBLOCAL – are missing from both programs. The obvious reason is that local objects and functions are never visible outside the file containing them; therefore, the local objects entry STOBJSTBLOCAL and local functions entry STTFUNSTBLOCAL are absent. Moreover, it is a well known fact that symbol table’s file object STTFILE does not exist in DSS [Symbol-Table, 2010]. The other fields represent the counts of elements in local, global and weak object categories. We can see in Figure 3.8 that these count values are totally different in both types of programs – the reason for high RA divergence.

Relocation Section. The RA divergence values for different relocation cate-
3. ELF-Miner: Using Structural Knowledge and Data Mining Methods To Detect New Malicious Executables

gories are shown in Figure 3.9. It is obvious that only 7 relocation types are used by both malware and benign files – RELASECTIONCOUNT, RELR38632, RELR386PC32, RELR386COPY, RELR386GLOBDAT, RELR386JMPSLOT and RELR386RELATIVE. A detailed analysis has revealed that RELR386PC32 is used only by malware programs. Note that RELR386PC32 relocation type supports the PC-relative addressing while RELR38632 supports absolute addressing. Some malware files use relative addressing while absolute addressing is commonly used by both benign and malware programs. Other .rel fields have high RA values that show their classification potential.

![Figure 3.9: Relocation section RA divergence graph for benign and malware files](image)

In .rela relocation section, the field RELASECTIONCOUNT represents the total number of .rela relocation types in a program. Three relocation categories – RELAR38632, RELAR386GLOBDAT and RELAR386JMPSLOT – are only used by benign programs. It is interesting to note that no malware writer has used .rela sections. As a result, the classification power of the information of .rela section is very high.

**Program Headers.** The frequency histogram of segment table is shown in Figure 3.10. Recall that we have included 8 segment headers in our features’ set. We have discussed their details in Section 3.3. We can easily conclude that most benign and malware programs do not contain processor specific semantic in program headers – the reason for low value of PT_LOPROC. In comparison, Segment PT_PHDR specifies the location and size of the program header table. It’s existence shows that the program header table is the part of program’s memory image that precedes the loadable segment entry. Most benign programs use this segment while most malware programs do not contain it.

We have proven our thesis that structural information – contained in the header of ELF files – provide good classification potential for discriminating malware programs from benign ones. Now we focus our attention to the classification methodology used in our ELF-Miner framework.

### 3.7 Classification Scheme

#### 3.7.1 Quantification

In Section 3.6, we have proven that the structural information of ELF files can be used to discriminate between malware and benign executables. We now use
3.7 Classification Scheme

Figure 3.10: Segments frequency histogram for benign and malware files

an information-theoretic measure – information gain – to rank different features in our datasets. As a result, we can visualize the patterns that exist in our malware datasets. Information gain measures the reduction in uncertainty if the values of an attribute are known [Cover and Thomas, 2006][Zhang and Tran, 2010]. For a given attribute $X$ and a class attribute $Y \in \{\text{Benign, Malware}\}$, the uncertainty is given by their respective entropies $H(X)$ and $H(Y)$. Then the information gain of $X$ with respect to $Y$ is given by $IG(Y; X)$:

$$IG(Y; X) = H(Y) - H(Y|X)$$

A higher value of information gain represents higher classification potential and vice versa. Note that information gain can vary from 1 to 0. We have created 8 datasets after combining benign files with each malware category – benign-virus, benign-worm, etc. We have used InfoGainAttributeEval attribute evaluator with Ranker search method in Wakanito Environment for Knowledge Acquisition (WEKA) [Witten and Frank, 2002]. Figure 3.11 shows the normal probability plot for information gain of features for Bkdrs, Cnstr, DoS, Expts + RK + HT, Fldrs, Trjns, Vrs + Spfr and Wrms. In Figures 3.11(a) and 3.11(b), two types of features can be observed: One type is composed of features with relatively large probability and the other type consists of features with relatively small existence probability. The majority of the features in all datasets have a very low value of information gain (less than 0.3), but some of them have information gain values as high as 0.82. It is interesting to see that a large number of features have relatively large existence probability but almost zero information gain. The reason behind the fact is that most of these features contain constant values in most of the files so their classification potential is zero. Another relevant observation is that the means of information gain distribution of malware datasets’ are 0.10, 0.03, 0.04, 0.11, 0.04, 0.03, 0.08, 0.04 for Bkdrs, Cnstr, DoS, Expts + RK + HT, Fldrs, Trjns, Vrs + Spfr and Wrms respectively. So we can envision – on the basis of information gain values – Cnstr and Trjns are the most difficult categories to classify for the classifiers that use higher information gain parameters to build classification rules. In the next section, this ranked (quantified) features’ set is then passed through preprocessing filters.
3. ELF-Miner: Using Structural Knowledge and Data Mining Methods To Detect New Malicious Executables

![Plot for the first 4 malware categories](image1.png) ![Plot for the Remaining malware categories](image2.png)

Figure 3.11: Probability distribution plot of Information Gain for features extracted from multiple malware datasets’

Table 3.3: Dataset analysis for redundant and useless features

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Instances</th>
<th>Features</th>
<th>Useless</th>
<th>Remaining</th>
<th>Useless (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bkdrs + Benign</td>
<td>931</td>
<td>383</td>
<td>182</td>
<td>201</td>
<td>47</td>
</tr>
<tr>
<td>Cnstr + Benign</td>
<td>750</td>
<td>383</td>
<td>188</td>
<td>195</td>
<td>49</td>
</tr>
<tr>
<td>DoS + Benign</td>
<td>768</td>
<td>383</td>
<td>196</td>
<td>187</td>
<td>41</td>
</tr>
<tr>
<td>Expts + RK + HT + Benign</td>
<td>987</td>
<td>383</td>
<td>190</td>
<td>193</td>
<td>49</td>
</tr>
<tr>
<td>Fldrs + Benign</td>
<td>768</td>
<td>383</td>
<td>196</td>
<td>188</td>
<td>49</td>
</tr>
<tr>
<td>Trjns + Benign</td>
<td>759</td>
<td>383</td>
<td>194</td>
<td>196</td>
<td>50</td>
</tr>
<tr>
<td>Vrs + Spfr + Benign</td>
<td>840</td>
<td>383</td>
<td>181</td>
<td>202</td>
<td>47</td>
</tr>
<tr>
<td>Wrms + Benign</td>
<td>778</td>
<td>383</td>
<td>196</td>
<td>187</td>
<td>49</td>
</tr>
</tbody>
</table>

3.7.2 Preprocessing Features

Remember in our ELF-Miner framework, we apply preprocessor filters to remove redundant and useless features – the features that have zero or very low classification potential. For classification purpose, we prepare different datasets by combining benign dataset with all malware datasets: Bkdrs + Benign, Cnstr + Benign, DoS + Benign, Expts + RK + HT + Benign, Fldrs + Benign, Trjns + Benign, Vrs + Spfr + Benign and Wrms + Benign – each of them having 931, 750, 768, 987, 758, 759, 840 and 778 instances respectively (see Table 3.3). In comparison, as mentioned before, we extract a fixed number of 383 features from an ELF file. Our forensic analysis has shown that ELF files usually do not contain all 34 section headers that are part of our complete features’ set (similar is the case of other headers). We have identified with the help of our forensic analysis and features’ set quantification that the empty fields that are assigned by default 0 values and fields with constant values have 0 classification potential. It is interesting to note in Table 3.3 that approximately 50% of features are redundant and useless. We have applied standard useless filter available in (WEKA) [Witten and Frank, 2002] to remove these features. It is clear from Table 3.3 that remaining features are actually given as input to a number of classification algorithms.

3.7.3 Classification

In the classification phase, our aim is to select a classifier that efficiently mines structural information extracted from ELF executables. The selected classifier must satisfy following requirements: (1) it must mine the structural information in real-time, (2) it must provide high detection accuracy, and (3) it must provide compre-
hensible and compact rules. In order to meet these requirements, we explore the design space along five dimensions: (1) evaluate different classification paradigms – classical and bio-inspired, (2) do a scalability analysis to rank features as a function of their classification potential, (3) do a robustness analysis of our system by randomly forging features of malware with that of benign files, (4) do a comprehensibility analysis of generated rules, and (5) perform a timing analysis to determine the realtime deployment feasibility of the system.

In our study, we have used supervised learning based classifiers [Hovsepian et al., 2010][Rodriguez-Gonzalez et al., 2010], four well known classical machine learning classifiers – RIPPER [Cohen, 1995], PART [Frank and Witten, 1998], C4.5 Rules [Quinlan, 1995] and J48 [Quinlan, 1993]. Similarly, we have used four well known bio-inspired genetic machine learning algorithms: (1) cAntMiner [Otero et al., 2008] is ant colony optimization based classifier, (2) XCS is a Michigan style learning classifier system [Wilson, 1995], (3) UCS is optimized for supervised learning environments [Bernadó-Mansilla and Garrell-Guiu, 2003], and (4) GAssist ADI is a Pittsburgh style learning classifier system [Bacardit and Garrell, 2007]. We skip details of classifiers in this chapter for brevity, but an interested reader may consult [Holland et al., 2000] for their description.

We have used the standard implementations of classical machine learning algorithms in (WEKA) [Witten and Frank, 2002]. Similarly, we have used implementations of bio-inspired evolutionary classifiers in Knowledge Extraction based on Evolutionary Learning (KEEL) [Alcala-Fdez et al., 2009] for our experiments. The objective of using standard tools is to remove any implementation related bias in our evaluation. Moreover, we have used the best configuration – determined after a number of plot studies – of a classifier in our experiments.

### 3.8 Experiments & Results

A stratified 10−fold cross validation procedure is followed for all experiments reported later in this section. In this procedure, we partition each dataset into 10 folds and 9 of them are used for training and the left over fold is used for testing. This process is repeated for all folds and the reported results are an average of all folds.

In our experiments, we consider malware detection as a two class problem – benign or malware detection. In such problems, the classification decision can possibly lie in one of the following categories: (1) True Positive (TP) is correct classification of a malicious ELF file as malicious, (2) True Negative (TN) is a benign ELF file instance classified as benign, (3) False Positive (FP) is misclassification of a benign ELF file as malicious, and (4) False Negative (FN) is misclassification of a malicious ELF file as benign. We define detection accuracy as following for our experiments:

\[
\text{DetectionAccuracy} = \frac{TP + TN}{TP + TN + FP + FN}
\]

**Detection Accuracy.** The results of our experiments are tabulated in Table 3.4. It is interesting to note that classical machine learning algorithms in general outperform bio-inspired evolutionary classifiers for our malware detection problem. Except XCS, other evolutionary classifiers provide comparable accuracy as that of
Table 3.4: ELF malware detection Avg. Accuracy and ADD comparison of evolutionary and non-evolutionary algorithms

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Evolutionary Classifiers</th>
<th>Non-Evolutionary Classifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C4.5-R</td>
<td>RIPPER</td>
</tr>
<tr>
<td>Bkdrs</td>
<td>100</td>
<td>0.991</td>
</tr>
<tr>
<td>±0.1</td>
<td>±0.2</td>
<td>±0.09</td>
</tr>
<tr>
<td>Cnstr</td>
<td>100</td>
<td>0.977</td>
</tr>
<tr>
<td>±0.1</td>
<td>±0.3</td>
<td>±0.04</td>
</tr>
<tr>
<td>DoS</td>
<td>100</td>
<td>0.911</td>
</tr>
<tr>
<td>±0.0</td>
<td>±0.4</td>
<td>±0.04</td>
</tr>
<tr>
<td>Expts</td>
<td>99.79</td>
<td>91.91</td>
</tr>
<tr>
<td>±0.004</td>
<td>±0.74</td>
<td>±0.04</td>
</tr>
<tr>
<td>Fltrs</td>
<td>100</td>
<td>0.766</td>
</tr>
<tr>
<td>±0.0</td>
<td>±0.34</td>
<td>±0.03</td>
</tr>
<tr>
<td>Trjns</td>
<td>99.96</td>
<td>76.28</td>
</tr>
<tr>
<td>±0.01</td>
<td>±0.38</td>
<td>±0.05</td>
</tr>
<tr>
<td>Vrs + Spfr</td>
<td>100</td>
<td>0.957</td>
</tr>
<tr>
<td>±0.0</td>
<td>±0.25</td>
<td>±0.1</td>
</tr>
<tr>
<td>Wrms</td>
<td>100</td>
<td>0.763</td>
</tr>
<tr>
<td>±0.0</td>
<td>±0.37</td>
<td>±0.05</td>
</tr>
</tbody>
</table>

Table 3.5: Comparison of ELF-Miner with other static analysis based techniques (AUC) discussed in Related Work section

<table>
<thead>
<tr>
<th>Datasets</th>
<th>IBK</th>
<th>JRIP</th>
<th>McBoost (C1)</th>
<th>KM</th>
<th>ELF-Miner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bkdrs</td>
<td>0.941</td>
<td>0.946</td>
<td>0.935</td>
<td>0.991</td>
<td>0.884</td>
</tr>
<tr>
<td>±0.003</td>
<td>±0.003</td>
<td>±0.003</td>
<td>±0.003</td>
<td>±0.003</td>
<td>±0.003</td>
</tr>
<tr>
<td>Cnstr</td>
<td>0.918</td>
<td>0.905</td>
<td>0.919</td>
<td>0.941</td>
<td>0.984</td>
</tr>
<tr>
<td>±0.003</td>
<td>±0.003</td>
<td>±0.003</td>
<td>±0.003</td>
<td>±0.003</td>
<td>±0.003</td>
</tr>
<tr>
<td>DoS</td>
<td>0.934</td>
<td>0.944</td>
<td>0.883</td>
<td>0.919</td>
<td>0.883</td>
</tr>
<tr>
<td>±0.003</td>
<td>±0.003</td>
<td>±0.003</td>
<td>±0.003</td>
<td>±0.003</td>
<td>±0.003</td>
</tr>
<tr>
<td>Expts</td>
<td>0.963</td>
<td>0.924</td>
<td>0.817</td>
<td>0.914</td>
<td>0.883</td>
</tr>
<tr>
<td>±0.003</td>
<td>±0.003</td>
<td>±0.003</td>
<td>±0.003</td>
<td>±0.003</td>
<td>±0.003</td>
</tr>
<tr>
<td>Fltrs</td>
<td>0.964</td>
<td>0.964</td>
<td>0.983</td>
<td>0.991</td>
<td>0.983</td>
</tr>
<tr>
<td>±0.003</td>
<td>±0.003</td>
<td>±0.003</td>
<td>±0.003</td>
<td>±0.003</td>
<td>±0.003</td>
</tr>
<tr>
<td>Trjns</td>
<td>0.943</td>
<td>0.946</td>
<td>0.964</td>
<td>0.952</td>
<td>0.964</td>
</tr>
<tr>
<td>±0.003</td>
<td>±0.003</td>
<td>±0.003</td>
<td>±0.003</td>
<td>±0.003</td>
<td>±0.003</td>
</tr>
<tr>
<td>Vrs + Spfr</td>
<td>0.934</td>
<td>0.944</td>
<td>0.924</td>
<td>0.931</td>
<td>0.944</td>
</tr>
<tr>
<td>±0.003</td>
<td>±0.003</td>
<td>±0.003</td>
<td>±0.003</td>
<td>±0.003</td>
<td>±0.003</td>
</tr>
<tr>
<td>Wrms</td>
<td>0.992</td>
<td>1</td>
<td>0.978</td>
<td>0.986</td>
<td>0.986</td>
</tr>
<tr>
<td>±0.002</td>
<td>±0.001</td>
<td>±0.001</td>
<td>±0.001</td>
<td>±0.001</td>
<td>±0.001</td>
</tr>
<tr>
<td>Avg. AUC</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

classical algorithms. RIPPER achieves a benchmark of 100% on our dataset that is closely followed by PART and J48.

**Average Detection Difficulty (ADD).** Another important question that we want to investigate: which malware category is the most difficult to detect? To answer this question, we take an average of detection accuracies of different classifiers and report average detection difficulty for each category at the end of each row in Table 3.4. The average detection difficulty of Bkdrs, Cnstr, DoS, Expts + HT + RK, Fltrs, Trjns, Vrs + Spfr and Wrms datasets’ are 96.49%, 96.80%, 97.34%, 97.17%, 96.56%, 96.55%, 97.0% and 96.54% respectively. The analysis of the results shows that Bkdrs, Wrms, Trjns and Fltrs are the most difficult malware categories to detect both for evolutionary and non-evolutionary classifiers. For our subsequent studies, we have short listed top 2 classifiers from Table 3.4 – RIPPER and J48.

**Comparison with other Static Analysis based techniques.** Table 3.5 tabulates the area under the ROC curves (AUCs) for ELF-Miner and two other static analysis based techniques McBoost and KT. We employ J48, Naive Bays, IBK and and RIPPER – two best classifiers for ELF-Miner and two best classifiers reported for other schemes [Kolter and Maloof, 2004]. Even a bird’s eye view of the table clearly demonstrate the superiority of ELF-Miner with more than 0.99 AUCs for most of the classifiers; hence, RIPPER achieves 1 AUC on all malware datasets. So far as, the results of other schemes are concerned, Kolter’s KM scheme provide the best results after our framework – i.e. from 0.94 to 0.98 AUC – using different classifiers; whereas, Mcboost schemes provide AUC between 0.90-0.93. It is notable...
that *Mcboost-C1* component – that deals with non-packed malicious executables – is functionally equal to its peer scheme *KM*. C2 component of *Mcboost* is for packed executables which is out of scope for us w.r.t his chapter.

**Scalability Analysis of Features’ Set.** Recall from Table 3.2 that our dataset consists of 734 instances of benign executables and 709 instances of different categories of malware executables. For scalability analysis of our features’ set, we have combined all benign and malware datasets’ and prepared a comprehensive dataset (i.e. 709+734=1443 instances). Afterwards, we rank all attributes using information gain measure. Finally, we select only those features that have an information gain of 0.1 or above; as a result, we get 88 top ranking features (see Table 3.6). Subsequently, we have created 9 different datasets by gradually removing the features that have information gain values greater than 0.89, 0.79, 0.69, 0.59, 0.49, 0.39, 0.29 and 0.19 respectively. The number of attributes in each category are shown in Table 3.6. The ROC curves in Figure 3.12 have been plotted by varying the threshold on output class probability [Fawcett, 2004], [Walter, 2005].

![Scalability analysis using J48](a) Scalability analysis using J48

![Scalability analysis using RIPPER](b) Scalability analysis using RIPPER

Figure 3.12: The magnified ROC plots for scalability analysis of ELF-Miner features’ set

It is interesting to see that our idea of using large number of features – extracted from different portions of ELF headers – has shown remarkable resilience, using any classifier, even when we just use 42 top ranked features. The system classifiers on the average merely show a 1 to 2% deterioration in true positive rate. However, the moment we remove the attributes having information gain more than 0.40, the true positive rate of classifiers significantly deteriorate by 1% to 11%.

**Robustness against Forged/Spoofed ELF Headers.** Now as a next step,
we analyze the "robustness" of our features’ set in a scenario when malware writers forge/spoof different sections of ELF header of their malware with that of benign files. We have gradually replaced malware headers with benign file headers and the corresponding true positive rate is given in Table 3.7. It is surprising to see that both classifiers are able to maintain 99% and 98% true positive rate, even when 77 of 88 features are forged/spoofed. It’s because, we have forged/spoofed randomly selected headers (mentioned in the first column of Table 3.7) and not necessarily the features that have high information gain values. This experiment validates that our features are “robust” against crafty attacks.

**Comprehensibility Analysis of Rules.** We now do the comprehensibility analysis of the generated rules by different classifiers. We have given some snapshots of generated rules by different classifiers in Table 3.8. It is interesting to see that in case of XCS and UCS, both classifiers have generated rules in the form of IF-THEN constructs by specifying intervals for all parameters in the dataset. As a result, the number of rules and their size becomes significantly large that makes it very difficult – if altogether not impossible – to comprehend and interpret them. In comparison, cAntMiner generates rules in IF-THEN form and GAssist-ADI generates rules in IF-THEN-ELSE form. It is interesting to note that both classifiers do not add all attributes in the antecedent part of the rules; as a result, the number and size of rules is significantly smaller compared with XCS and UCS that increases the comprehensibility and interpretability of their rules.

In case of classical machine learning algorithms, RIPPER, PART and C4.5 Rules generate compact and simple rules in the form of IF-THEN-ELSE. They also highlight the number of data samples covered by a specific rule. These classifiers do not add all input parameters in the antecedent part of the rules. Consequently, their comprehensibility is high. J48 is the only classifier that builds complex hierarchical rules. Its rules are accurate but complex as compared with other classifiers. A closer look at the rules confirm our forensic analysis in Section 3.6: the classification power of features of relocation section (.rela) is very high followed by the information from different section headers – .comment, .note, .strtab, .symtab, and .sbss.

**Timing Analysis.** We now analyze the processing overhead of ELF-Miner that includes time for features extraction, preprocessing, classifier training and testing. On an average, features extraction and preprocessing times per instance are 15.391 and 0.96 milliseconds respectively. The training times of J48 and RIPPER are 0.173 and 0.616 milliseconds per instance respectively. Similarly, the testing time of both J48 and RIPPER is 6.9 microseconds. So overall processing overhead of ELF-Miner is approximately 16 to 17 milliseconds per instance (file) using any of the two best classifiers. Such a small processing overhead makes our ELF-Miner

<table>
<thead>
<tr>
<th>Headers Removed</th>
<th>Removed Attr.</th>
<th>Remaining Attr.</th>
<th>J48 Results (%)</th>
<th>RIPPER Results (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Accuracy</td>
<td>TP Rate</td>
</tr>
<tr>
<td>Note, GOT, SBSS, PLT</td>
<td>0</td>
<td>88</td>
<td>100</td>
<td>0.0</td>
</tr>
<tr>
<td>Reloc, DynSym</td>
<td>18</td>
<td>70</td>
<td>99.45</td>
<td>0.4</td>
</tr>
<tr>
<td>ELF, Cmnt, BSS, DynSec</td>
<td>10</td>
<td>60</td>
<td>99.24</td>
<td>0.4</td>
</tr>
<tr>
<td>Symbolic Sec</td>
<td>12</td>
<td>47</td>
<td>99.51</td>
<td>0.3</td>
</tr>
<tr>
<td>[DynStr, DynSym, Fini, Hash, Init]SecH</td>
<td>7</td>
<td>40</td>
<td>99.44</td>
<td>0.4</td>
</tr>
<tr>
<td>[StrTab, SymTab]SecH</td>
<td>10</td>
<td>19</td>
<td>99.64</td>
<td>0.4</td>
</tr>
<tr>
<td>[GOTPLT, RELDyn]SecH</td>
<td>11</td>
<td>10</td>
<td>99.55</td>
<td>0.4</td>
</tr>
</tbody>
</table>

Table 3.7: Malware detection capability of ELF features’ set with spoofed headers
Table 3.8: Comprehensibility analysis of rules generated by all evolutionary and non-evolutionary classifiers

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF RELDYNSHENTSIZ[$&gt; 8$] AND STRTABSHTYPE[$&lt; 3$]: benign</td>
</tr>
<tr>
<td></td>
<td>IF RELAR386JMPSLOT[$&lt; 5$]: malware</td>
</tr>
<tr>
<td>XCS</td>
<td>IF ELFHEININDENT[$[0.0 0.5]$] AND ELFHETYPE[$[0.0 0.52]$] ... HASHTABLESIZE[$[0.0 0.06]$]: malware</td>
</tr>
<tr>
<td></td>
<td>IF ELFHEININDENT[$[0.0 0.8]$] AND ELFHETYPE[$[0.0 0.7]$] ... HASHTABLESIZE[$[0.0 0.35]$]: malware</td>
</tr>
<tr>
<td>UCS</td>
<td>IF ELFHEININDENT[$[0.0 1.0]$] AND ELFHETYPE[$[0.0 1.0]$] ... HASHTABLESIZE[$[0.0 1.0]$]: benign</td>
</tr>
<tr>
<td></td>
<td>IF ELFHEININDENT[$[0.0 1.0]$] AND ELFHETYPE[$[0.0 1.0]$] ... HASHTABLESIZE[$[0.0 0.9]$]: malware</td>
</tr>
<tr>
<td>GAassist-ADI</td>
<td>IF RELAR386JMPSLOT[$&gt; 0$] [661/661]: benign</td>
</tr>
<tr>
<td></td>
<td>ELSE IF RELAR386JMPSLOT[$&lt; 0$] [177/177]: malware</td>
</tr>
<tr>
<td></td>
<td>ELSE benign</td>
</tr>
<tr>
<td>C4.5 Rules</td>
<td>IF RELAR386JMPSLOT[$&gt; 0$] [661/661]: benign</td>
</tr>
<tr>
<td></td>
<td>ELSE IF RELAR386JMPSLOT[$&lt; 0$] [177/177]: malware</td>
</tr>
<tr>
<td></td>
<td>ELSE IF RELAR386JMPSLOT[$&lt; 11.0$] AND BSSSHTYPE[$&gt; 1.0$] [23/24]: malware</td>
</tr>
<tr>
<td></td>
<td>ELSE benign</td>
</tr>
<tr>
<td>RIPPER</td>
<td>IF RELAR386JMPSLOT[$&gt; 0$] [661/661]: benign</td>
</tr>
<tr>
<td></td>
<td>ELSE malware</td>
</tr>
<tr>
<td></td>
<td>IF RELAR386COPY[$&gt; 10$] [659/660]: benign</td>
</tr>
<tr>
<td></td>
<td>ELSE IF COMENTSHTYPE[$&gt; 0$] [24/225]: malware</td>
</tr>
<tr>
<td></td>
<td>ELSE benign</td>
</tr>
<tr>
<td>PART</td>
<td>IF RELAR386JMPSLOT[$&gt; 0$] [660/660]: benign</td>
</tr>
<tr>
<td></td>
<td>ELSE malware</td>
</tr>
<tr>
<td>J48</td>
<td>IF GOTPLTSHYPE[$&lt; 0$] ... IF RELAR386GLOBDAT[$&lt; 247$]: malware</td>
</tr>
<tr>
<td></td>
<td>... ELSE IF RELAR386GLOBDAT[$&gt; 247$]: malware</td>
</tr>
<tr>
<td></td>
<td>... IF COMENTSHTYPE[$&lt; 0$]: benign</td>
</tr>
<tr>
<td></td>
<td>... ELSE IF COMENTSHTYPE[$&gt; 0$]: malware</td>
</tr>
<tr>
<td></td>
<td>ELSE IF GOTPLTSHYPE[$&gt; 0$] ... IF SYMSTTHIPROC[$&lt; 19$]: malware</td>
</tr>
<tr>
<td></td>
<td>... ELSE IF RELAR386RELATIVE[$&lt; 3$]: malware</td>
</tr>
<tr>
<td></td>
<td>... ELSE IF RELAR386RELATIVE[$&gt; 3$]: benign</td>
</tr>
<tr>
<td></td>
<td>... ELSE IF SYMSTTHIPROC[$&gt; 19$]: benign</td>
</tr>
</tbody>
</table>

3.9 Conclusions

In this chapter, we have introduced ELF-Miner, a malware detection framework that mines structural features – extracted from different sections of ELF headers – of Linux executables. We have done detailed forensic analysis to investigate the fields that have high classification potential to detect malicious executables. Our system achieves more than 99.9% accuracy depending on the selected classifier. We have also done scalability experiments to show that ELF-Miner is able to provide high detection accuracy even with a small number of features. We have also shown that forging the features does not result in significantly degrading the detection accuracy. It will be an interesting future work to put efforts to establish the generality of our approach on executables of other operating systems – Mac OS X, Symbian etc.
### Description of ELF Sections

<table>
<thead>
<tr>
<th>No</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.text</td>
<td>Executable instructions</td>
</tr>
<tr>
<td>2</td>
<td>.bss</td>
<td>Uninitialized data in program image</td>
</tr>
<tr>
<td>3</td>
<td>.comment</td>
<td>Version control information</td>
</tr>
<tr>
<td>4</td>
<td>.data</td>
<td>Initialized data variables in image</td>
</tr>
<tr>
<td>5</td>
<td>.data1</td>
<td>Initialized data variables in image</td>
</tr>
<tr>
<td>6</td>
<td>.debug</td>
<td>Program debug symbolic information</td>
</tr>
<tr>
<td>7</td>
<td>.dynamic</td>
<td>Dynamic linking information</td>
</tr>
<tr>
<td>8</td>
<td>.dynstr</td>
<td>Dynamic string section</td>
</tr>
<tr>
<td>9</td>
<td>.dynsym</td>
<td>Dynamic symbol information</td>
</tr>
<tr>
<td>10</td>
<td>.fini</td>
<td>Process termination code</td>
</tr>
<tr>
<td>11</td>
<td>.hash</td>
<td>Hash table</td>
</tr>
<tr>
<td>12</td>
<td>.init</td>
<td>Process initialization code</td>
</tr>
<tr>
<td>13</td>
<td>.got</td>
<td>Global offset table</td>
</tr>
<tr>
<td>14</td>
<td>.interp</td>
<td>Path name for a program interpreter</td>
</tr>
<tr>
<td>15</td>
<td>.line</td>
<td>Line number information of symbolic .debug</td>
</tr>
<tr>
<td>16</td>
<td>.note</td>
<td>File notes</td>
</tr>
<tr>
<td>17</td>
<td>.plt</td>
<td>Procedure link table</td>
</tr>
<tr>
<td>18</td>
<td>.rodata</td>
<td>Read only data</td>
</tr>
<tr>
<td>19</td>
<td>.rodata1</td>
<td>Read only data</td>
</tr>
<tr>
<td>20</td>
<td>.shstrtab</td>
<td>Section header string table</td>
</tr>
<tr>
<td>21</td>
<td>.strtab</td>
<td>String table</td>
</tr>
<tr>
<td>22</td>
<td>.symtab</td>
<td>Symbol table</td>
</tr>
<tr>
<td>23</td>
<td>.sdata</td>
<td>Initialized non-const global and static data</td>
</tr>
<tr>
<td>24</td>
<td>.sbss</td>
<td>Static better save space</td>
</tr>
<tr>
<td>25</td>
<td>.lit8</td>
<td>8-byte literal pool</td>
</tr>
<tr>
<td>26</td>
<td>.gptab</td>
<td>Size criteria info for placement of data items in the .sdata</td>
</tr>
<tr>
<td>27</td>
<td>.conflict</td>
<td>Additional dynamic linking information</td>
</tr>
<tr>
<td>28</td>
<td>.tdesc</td>
<td>Targets description</td>
</tr>
<tr>
<td>29</td>
<td>.lit4</td>
<td>4-byte literal pool</td>
</tr>
<tr>
<td>30</td>
<td>.reginfo</td>
<td>Information about general purpose registers for assembly file</td>
</tr>
<tr>
<td>31</td>
<td>.liblist</td>
<td>Shared library dependency list</td>
</tr>
<tr>
<td>32</td>
<td>.rel.dyn</td>
<td>Runtime relocation information</td>
</tr>
<tr>
<td>33</td>
<td>.rel.plt</td>
<td>Relocation information for PLT</td>
</tr>
<tr>
<td>34</td>
<td>.got.plt</td>
<td>Holds read-only portion of global Offset Table</td>
</tr>
</tbody>
</table>
Chapter 4

In-Execution Dynamic Malware Analysis and Detection by Mining Information in Process Control Blocks of Linux OS

4.1 Introduction

Malware detection schemes based upon static analysis of source code & executable are prone to code obfuscation and polymorphism. Therefore, runtime behavior of processes running on an end-host is being actively used to dynamically detect malware. Most of these detection schemes build model of run-time behavior of a process on the basis of its data flow and/or sequence of system calls. The major shortcomings of these schemes are their higher processing overheads and system call mimicry attacks. In view of this, we present a novel and lightweight malware detection scheme based on inexecution dynamic analysis of process control blocks of Linux kernel. To the best of our knowledge, nobody before us used PCB for malware detection.

Computer malware is becoming an increasingly significant threat to the computer systems and networks world-wide. In recent years, security experts have observed a massive increase in the number and sophistication of new malware. According to a recent threat report by Symantec, in 2008 alone, 5491 new software vulnerabilities were reported, 1.6 million new malware signatures were created, 245 million new attacks were reported, and the financial losses caused by malware soared to more than 1 trillion dollars [Fossi, 2010]. It is, however, interesting to note that though 50% of new malware are simply repacked versions of known malware [Taha, 2007], even then they successfully evade existing commercial-off-the-shelf antivirus software (COTS AV) because they utilize static signature-based techniques [Szor, 2005] that are not robust to code obfuscation and polymorphism.

To overcome shortcomings of existing COTS AV, malware researchers are focusing their attention to non-signature behavior based intelligent detection techniques that can detect new malware on zero day (at the time of its launch). Such techniques can be broadly classified into two categories: (1) static, and (2) dynamic. The main objective of static behavior based techniques is to analyze the informa-
4. In-Execution Dynamic Malware Analysis and Detection by Mining Information in Process Control Blocks of Linux OS

In comparison, dynamic detection techniques try to model run-time behavior of a process; as a result, they cannot be evaded by mere code obfuscation or polymorphism. Two types of well known dynamic schemes have been proposed: (1) system call sequencing, and (2) flow graphs. The system call sequences based techniques (presented in [Ahmed et al., 2009] [Bayer et al., 2006][Willems et al., 2007] [Yin et al., 2007][Christodorescu et al., 2005] and [Li et al., 2008]) build the behavioral model on the basis of the sequence of invoked system calls and the information passed in their arguments. These techniques have four well known shortcomings: (1) processing overhead of logging system calls, (2) high false alarm rate, (3) ability to make a decision only after analyzing the complete trace of the execution sequence of a process (by that time a malicious process has already done the damage), and (4) they can be easily evaded by simply reordering the system calls or adding irrelevant system calls to invalidate the sequence that represents malware.

The basic concept behind flow graph techniques is to construct taint graph of a running process by analyzing its data flow model [Yin et al., 2007][Kirda et al., 2006]. The graph provides valuable insight about the activities of a running process. Most of the proposed schemes require a virtual machine with a shadow memory to keep track of taint information. A related system is NICKLE [Riley et al., 2008], but it has been demonstrated that return oriented rootkits [Hund et al., 2009] can evade it. These schemes – like their system call counterparts – have high false alarm rate, high processing overheads and high detection time (sometimes of the order of minutes). It is rightly concluded in a recent paper [Kolbitsch et al., 2009] that most of the above-mentioned dynamic techniques are novel and promising but still a leap jump is required in their detection accuracy, detection time, and processing overheads to establish their effectiveness to replace or complement traditional anti-virus software at an end host.

To meet these challenges, a novel technique based on dynamic analysis is proposed that uses genetic footprint of a running process. The thesis of using genetic footprint is: the state diagram of malicious processes should be different from that of benign processes because of difference in their activities. The genetic footprint is defined as a set of 16 out of 118 parameters, which are maintained by the kernel of an operating system, in the PCB of a process\(^1\), to keep track of the state of an executable process \(^2\). To be more specific, malicious processes that try to steal or corrupt data and information – like passwords, keystrokes, files etc. – try to covertly capture the information that is not intended for them [Kolbitsch et al., 2009] without a user’s consent. In comparison, the benign processes follow access control policies to acquire the legitimate information. Similarly, the state of a malicious process

\(^1\)Throughout this chapter, we will use the terms PCB, task structure, and task_struct interchangeably.

\(^2\)The feasibility of using parameters of “task structure”, maintained in the kernel of Linux, is investigated by the authors in a preliminary pilot study reported in [Shahzad et al., 2011a].
that tries to replicate itself on storage media or network interfaces will be different from the benign process that runs only once, performs its given tasks and exits.

As mentioned before, the genetic footprint is maintained in the “task structure” by the kernel of every operating system. Linux operating system is selected because of its open source advantage that provides fine grain control over kernel structures; as a result, systematic studies/evaluations are conducted on them. In case of Linux, the information about the current state of a process is maintained in the task_struct structure. The aim is to analyze the temporal behavior of the genetic footprint; therefore, a system call is developed that dumps 118 fields of task_struct structure for 15 seconds with a resolution of one millisecond (ms). This time interval is reconfigurable and at the moment it is based on the observation that most of the malicious processes in the selected dataset either finish their intended activity within this time duration or at least start showing a distinct behavior to characterize them as benign or malicious.

The proposed scheme is evaluated on a dataset that consists of 105 benign processes and 114 recently collected malware processes of Linux. The results of the experiments prove that proposed scheme achieves a detection accuracy of 96% with 0% false alarm rate. Moreover, it is able to detect a malicious process in less than 100 ms from “the start of a malicious activity”. It is emphasized the use of “the start of a malicious activity” instead of “the start of a malicious process” because the use of genetic footprint of a process enables the detection of even those malicious processes which mostly behave like benign processes and perform malicious activity for a very short duration somewhere in the middle (such as backdoors). Most of existing run-time behavior analysis techniques fail to detect such malicious processes. Also to the best of our knowledge, no existing runtime analysis technique is able to detect a malicious activity in less than 100 ms. A robustness analysis of the proposed technique is also presented in circumstances when a crafty attacker splices the genetic footprint of a benign process with that of a malicious process at different positions. The results of experiments show that the proposed scheme is robust to such evasion attempts. It is also demonstrated that it is difficult to evade proposed technique if it is used in conjunction with recently proposed techniques – discussed in Section 4.6.

To conclude, the major contributions of the work presented in this chapter are: (1) architecture of the proposed dynamic malware detection framework based upon the concept of genetic footprint – a set of selected parameters (maintained inside the task structure of a kernel of an operating system for each running process) – that defines the semantics and behavior of an executing process, and (2) a testing and validation framework to prove that our framework has the capability to not only detect a running malware within 100 ms of the start of a malicious activity but it also provides 96% detection accuracy with a 0% false alarm rate.

The rest of the chapter is organized as follows. Section 4.2 builds the motivation for using information in the task structure by doing a time series forensic study of selected malware and benign processes. In Section 4.3, the characteristic of dataset are discussed that is used for experiments. Section 4.4 explains in detail the functionality of three modules of the proposed scheme: (1) features logging, (2) features selection, and (3) classification. The design of the scheme is based on systematic investigation and evaluations. Section 4.5 describes the results of experiments and in doing so intriguing insights are provided about the behavior of the proposed
4.2 Forensic Analysis of Benign and Malicious Processes

The intuition is systematically built in order to answer a tricky question: why genetic footprint of a malware process should be different from that of a benign one? The forensic investigation is conducted to analyze the execution behavior of benign and malware processes. Both the benign and malware processes are selected from different categories. The benign processes belong to text editors, image utilities and system utilities categories; while malware processes belong to trojan horses, networms, and virus categories. A brief description of each type of process is provided that will help in correlating their behavior with their genetic footprint.

4.2.1 Benign Processes

**hwclock (CLK).** This utility accesses the hardware clock of the system. It displays the current time and is responsible for synchronizing the system and hardware clocks periodically.

**ED.** It is a command line text editor that can create, display, and modify text files. When it is invoked with a filename, it copies the text of original file into its buffer in a temporary file and all subsequent changes are made to this file and are written into the original file once explicitly saved. This utility is invoked with a text file but the user does not edit the file for 15 seconds.

**ppmtoterm (PPMT).** This is a command-line image utility that converts a .ppm image to ANSI ISO 6429 ASCII image and displays it on the terminal of Linux OS. It performs color approximations, measures the minimum cartesian distance between RGB vectors and finally generates palettes. Unlike the text editor **ED**, it does all above-mentioned steps in an automated fashion.

4.2.2 Malware Processes

**Linux.Backdoor.Kaiten.a (TKTN).** It is a trojan horse [Symantec, 2010a] that opens a backdoor on the victim computer and uses an IRC client to connect to IRC servers on the port TCP 6667. It connects to a predetermined IRC channel and listens for commands, which are then used to perform malicious activities on the victim host. It performs the following malicious activities: (1) launches DDOS attacks using SYN and UDP packets, (2) downloads and executes remote scripts and files, (3) changes a client’s nickname, (4) changes servers, (5) spoofs an IP address, (6) kills running processes, (7) generates floods, and (8) changes system files: `/etc/rc.d/rc.local` and `/etc/rc.conf`.

**Net-Worm.Linux.Sorso-b4264 (WSRO).** It is a Linux worm that infects the samba server [Symantec, 2010b]. It establishes a connection and sends exploit code...
4.2 Forensic Analysis of Benign and Malicious Processes

to the shell on the server. Once the server runs the exploit code, it in turn downloads malicious executables that try to infect more samba servers, replace the http daemon with the hijacked one. Moreover, it hides its created process, steals IP addresses and sends them to remote hosts to perform various actions enlisted in [Symantec, 2010b].

**Virus.Linux.Satyr.a (VSTR).** It is a non-memory resident parasitic Linux virus [Virus-List, 2010] that has multiple aliases. Its main task is to look for other executable files in the system and infect them. The virus infects files in the directories and it shifts down the contents of a victims’ file and writes itself in the file header. To release control to the host file, the virus “disinfects” it to a temporary file and then executes it. The virus does not manifest itself in anyway but its body contains a copyright text string.

![Figures](https://via.placeholder.com/150)

(a) Number of page frames owned by benign processes  
(b) In-volunteer context switches of benign processes  
(c) Volunteer context switches of benign processes  
(d) Number of page frames owned by malicious processes  
(e) In-volunteer context switches of malicious processes  
(f) Volunteer context switches of malicious processes

**Figure 4.1:** Forensic analysis of benign and malicious processes

**Virus.Linux.ELF_X23 (VX23).** Once ELF_X23 [TrendMicro, 2010] executes, it infects other ELF files in the current directory. It checks if the file has a .X23 extension and whether it is an executable in the user group. If it does not have this extension and if it is an executable in the user group, it concatenates the .X23 extension to the host file name. This serves as a copy of the original host file. After this, it copies its virus code to the original host file. Finally, it changes the attribute of the file to readable, writable, and executable for all user groups. Once the virus is finished, it transfers control back to the copy of the original file with the .X23 extension.

### 4.2.3 Forensic Analysis

The forensic analysis is presented on three parameters of Linux `task_struct` structure to build an insight into the execution behavior of benign and malicious processes. The aim is to get an understanding that how the execution behavior of a
process is correlated with the task_struct parameters. The sample parameters are: (1) the number of page frames owned by a process, (2) the number of in-volunteer context switches of a process, and (3) the number of volunteer context switches of a process. A time series analysis of these processes is done. In order to show distinct patterns, only two processes are shown in a figure. (It is experienced that once the parameters of four processes are plotted in a single figure, the legibility of the figure is severely impaired.)

**Page frames owned by a process.** It is interesting to note in Figures 4.1(a) and 4.1(d) that benign programs allocate memory in the beginning of their execution and then their memory usage does not change in small intervals of milliseconds. A malware, on the other hand, wants to cause the damage as quickly as possible; therefore, its memory usage shows oscillations. (Note that in the selected dataset, 3 malware samples finish in less than 10 ms; while 11 finish in less than 30 ms.) Moreover, benign processes – CLK and ED – show a uniform memory usage pattern in small intervals while VX23 malware shows an oscillatory behavior. The moment WSRO starts the malicious activity, its memory usage also changes in small intervals of time.

**Volunteer and In-volunteer context switching.** A volunteer context switch means that a processes relinquishes control of the processor before expiry of its allocated time quantum. On the other hand, if a scheduler intervenes to stop a running process at the expiry of its allocated time quantum, it is called an in-volunteer context switch.

It is again interesting to see in Figures 4.1(b) and 4.1(c) that benign processes mostly execute in a non-greedy fashion; consequently, they have an increasing number of volunteer context switches and near zero in-volunteer context switches. On the other hand, it can be seen in Figures 4.1(e) and 4.1(f) that the malicious processes mostly operate in a greedy mode; as a result, they are preempted by the scheduler resulting in an increasing number of in-volunteer context switches and near zero volunteer context switches. The above analysis provides a preliminary insight that Linux task_struct can be used to distinguish between benign and malicious processes.

To generalize the above mentioned assertion, the histograms of three task structure parameters is plotted of 20 benign and malware processes each in Figure 4.2. (Note that the processes are randomly selected from the chosen dataset.) From Figures 4.2(a) and 4.2(b), it is concluded that hiwater rss plots of benign processes have large spreads, implying continuous memory allocation. In contrast, malicious processes show sparsely located spikes for hiwater rss depicting that the allocated memory remains constant during most of the execution trace. Furthermore, hiwater rss spans over a shorter domain (i.e. 0-250) for malicious processes as compared to the benign ones (i.e. 0-2000). (Frequency and hiwater rss axis are magnified for didactic purpose.)

Similarly, Figures 4.2(c) to 4.2(f) show voluntary and involuntary context switching behavior of benign and malicious processes. It is visible in both figures that benign processes are more likely to switch their context voluntarily as compared to malicious processes. The same behavior can be inferred from Figures 4.2(e) and 4.2(f), where it is visible that benign processes are rarely preempted by the OS scheduler. Since malicious processes have higher predilection to hold CPU; therefore, they are more frequently preempted. Also, it can be observed that malicious
processes mostly finish quickly (in less than 200 ms) and the majority of them in less than 50 context switches. In comparison, the majority of benign processes execute till 2000 context switches (or even more).

It can be concluded that the forensic and histogram analysis both ascertain the ground truth: *some parameters of task_struct of Linux have the potential to characterize a process as a benign or malicious.*
4.3 Dataset

In section 4.2, the fact is established that Linux task_struct parameters can be used to discriminate between benign and malicious processes. Now a larger dataset of benign and malicious processes is taken for a detailed analysis of more task_struct parameters and their classification potential.

In this section, the dataset and the methodology is presented to log parameters of task structure after every 1 ms. 114 malware and 105 benign samples are used for this purpose. The statistics of all these samples are tabulated in Table 4.1.

In Table 4.1, it can be seen that benign samples are divided into four major categories: games (Gms), image utilities (ImgUtl), text editing utilities & converters (TEd), and miscellaneous Linux shell commands (SCmd). These benign files are collected from Linux operating system’s directories /bin, /sbin, and /usr/bin. The diversity in the benign dataset is ensured by selecting files of different sizes – ranging between 10 KiloBytes (KB) to 2 MegaByte (MB) – and categories from above-mentioned directories.

Malware samples have been collected from “Offensive Computing” [Offensive-Computing, 2009] and “VX Heavens” [VX-Heavens, 2009] malware collections. These 114 malware samples can be divided into 8 different malware categories – Exploits (Exp), Flooders (Fldrs), Net-Worms (NWrm), Rootkits + Hacktools (Rkt), Backdoors (BkDr), Trojans (Trjn), and Virus (Vrs). Malware processes execute in an automated manner and usually don’t take input from the user while benign processes consist of a variety of execution patterns; automated (system processes), semi-automated (take users’ input only once e.g. image converters) and manually operative (interactive and non-interactive text editors, firefox etc). Both the malware and benign processes have been dumped in almost all execution states. In order to have a balanced dataset, it is ensured that the percentage of benign and malicious files in a specific category (based upon size) is approximately the same. For an interested reader, the complete list of 114 malware and 105 benign processes is provided in Table 4.6.

Table 4.1: Benign and malware file size distribution

<table>
<thead>
<tr>
<th>Size</th>
<th>SCmd</th>
<th>ImgUtl</th>
<th>TEd</th>
<th>Gms</th>
<th>All</th>
<th>Exp</th>
<th>Fldrs</th>
<th>NWrm</th>
<th>Rkt</th>
<th>BkDr</th>
<th>Trjn</th>
<th>Vrs</th>
<th>All</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;10K</td>
<td>6</td>
<td>19</td>
<td>1</td>
<td>0</td>
<td>26</td>
<td>25</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>19</td>
<td>8</td>
</tr>
<tr>
<td>10-50K</td>
<td>35</td>
<td>9</td>
<td>3</td>
<td>4</td>
<td>51</td>
<td>48.5</td>
<td>5</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>9</td>
<td>14</td>
<td>22</td>
<td>65</td>
</tr>
<tr>
<td>50-100K</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>10</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>7</td>
<td>0</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>100-500K</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>17</td>
<td>14</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>500K-2048K</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>28</td>
<td>22</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td>60</td>
<td>28</td>
<td>8</td>
<td>9</td>
<td>105</td>
<td>100</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>28</td>
<td>17</td>
<td>41</td>
<td>114</td>
<td>100</td>
</tr>
</tbody>
</table>

4.4 Architecture of Dynamic Malware Detection Framework

In this section, the architecture of proposed scheme is presented that consists of three modules: (1) features logger, (2) features analyzer, and (3) classifier. In the following subsections a methodology is presented to select the features that define the genetic footprint. Moreover, a systematic analysis is performed on features’ set to select a suitable machine learning classifier. The architecture of the proposed framework is shown in Figure 4.3. Each module is discussed individually.
4.4 Architecture of Dynamic Malware Detection Framework

Figure 4.3: Block diagram of the dynamic malware detection framework

4.4.1 Features Logger

The job of features logger is to periodically dump 118 fields of task_struct structure of Linux. Some of the most notable parameters are number of page frames, volunteer and in-volunteer context switches, number of page table locks, number of page faults, virtual memory used, CPU time in system, mode of a process, number of page tables etc. The parameters are logged using customized kernel system call framework that invokes customized system calls to extract relevant information from task_struct structure of a desired process after every millisecond for 15 seconds. As system calls are endemic part of kernel, so these can access kernel structures directly. Linux kernel stores the state of processes in a doubly circular linked list of task_struct structure and maintains a global variable of this structure named – current – that provides access to the task structure of the current process. Next or previous processes can be accessed using next or previous members of current. In this way, the state of any process can be accessed by moving forward or backward in this circular linked list. A customized system call tracks the process under consideration by its name in the circular list of Linux kernel. As soon as it finds the process, it logs the fields of task_struct in a separate data file with the same name as of the running process. Through this system call, a maximum of 15000 samples for each process have been dumped. The processes that finish earlier, of course have the number of samples corresponding to their total execution time. This methodology is used to dump the task structure parameters for 105 benign and 114 malicious processes.

4.4.2 Features Analyzer

In order to short-list features with high classification potential, feature preprocessing has been done in two steps: (1) identifying and filtering the fields that are not relevant to the behavior of a process, and (2) the time series analysis of remaining fields is performed to identify features with a high classification potential. The fields that do not depend on the behavior of a process are constant fields, process identifiers, bit combinations (flags) etc; therefore, it is important to remove them from the features’ set so that they do not misguide a classifier during the training process. In case of Linux, 23 parameters are various kinds of offsets, 9 are flags, and 50 are either constants or zeros that show no discriminating behavior. Once the first step is finished, only 36 parameters are left on which the second step is performed.

In the second step, the time series mean of each parameter for all benign and
4. In-Execution Dynamic Malware Analysis and Detection by Mining Information in Process Control Blocks of Linux OS

Figure 4.4: Plots of various statistical aspects of short listed parameters.
malicious files is calculated using the following equation:

\[ \mu_{i,k} = \frac{1}{N} \sum_{j=1}^{N} x_{i,j,k} \quad i = 1 \rightarrow t_x, k = 1 \rightarrow 36, \quad (4.1) \]

where \( i \) represents the time instance, \( t_x \) is the termination time of a process (in milliseconds) or 15000 which ever is smaller, \( j \) represents the processes’ identification number, \( N = 105 \) for benign and \( N = 114 \) for malicious processes. \( \mu_{i,k} \in T_k \), where \( T_k \) are the time series means of parameters \( k = 1 \) to 36 (remember in this step only 36 parameters are processed).

The time series means of these 36 fields of benign and malicious processes help in short listing the fields that are different in benign and malicious processes. Moreover, it is also equally important that the selected parameter should have low variance in all samples of benign and malware processes; otherwise, it would lead to high false positives and false negatives respectively. In order to get an understanding about the spread of a feature, its variation is computed using the following equation:

\[ \sigma^2_{i,k} = \frac{1}{N} \sum_{j=1}^{N} (x_{i,j,k} - \mu_{i,k})^2 \quad i = 1 \rightarrow t_x, k = 1 \rightarrow 36 \quad (4.2) \]

where \( i, j, N, t_x \) and \( k \) represent the same quantities as in Eq. (4.1). \( \sigma^2_{i,k} \in V_k \), where \( V_k \) are the time series variances of parameter \( k \). In order to factor out the impact of the value of a feature, a coefficient of variance is defined. The coefficient of variance of a feature can be computed using the following equation:

\[ cv_{i,k} = \frac{\sigma_{i,k}}{\mu_{i,k}} \quad i = 1 \rightarrow t_x, k = 1 \rightarrow 36 \quad (4.3) \]

where \( i, j, \mu, \sigma, t_x \) and \( k \) represent the same quantities as in Eqs. (4.1) and (4.2). \( cv_{i,k} \) represents the coefficient of variance of parameter \( k \) at time instant \( i \). If the coefficient of variation approaches 1, it means that the standard deviation of a parameter is approximately the same as its mean. If a feature has a large coefficient of variance, intuitively speaking it is not a good idea to use it because it will confuse the learning process that ultimately would result in reducing the detection accuracy.

Before going into the details of the parameter selection procedure, a formal definition of the genetic footprint is provided. Let \( Y_k \) be a time series of parameter \( k \) that terminates at time instant \( n \), then \( Y_k \) is given by:

\[ Y_k = \{y|y = (t_{i,k}, x_{i,k}) \land i = [0, n]\} \quad (4.4) \]

where \((t_{i,k}, x_{i,k})\) is the ordered pair which represents the value \( x_{i,k} \) of the time series \( Y_k \) at time instant \( t_i \). The termination time of the process is defined by the symbol \( t_x \). In this scenario, \( n = t_x \) if \( t_x < 15000 \) and \( n = 15000 \) if \( t_x \geq 15000 \). In terms of its parameters, the task structure \( T \) is defined as:

\[ T = \{Y_k|y \in Y_k \text{ is a parameter of Linux task_struct}\} \quad (4.5) \]

This equation means that the \( T \) is basically a set of time series of the parameters of the task_struct structure. The genetic footprint \( \mathbb{G} \) of a process, is essentially a
subset of the set $T$ i.e. $G \subset T$ whose elements are time series of those parameters that have certain required properties. Formally, $G$ is defined in the following equation:

$$G = \{Y_k \mid \forall Y_k \in T \Rightarrow |\mu_b(Y_k) - \mu_m(Y_k)| > \epsilon \land 0 < cv_{b,m}(Y_k) < 3 \land |Z| \geq |Y_k| \times 0.9545\}$$  \hspace{1cm} (4.6)$$

where $\mu_b(Y_k)$ is the time series mean of the time series $Y_k$ of benign processes and $\mu_m(Y_k)$ shows the time series mean of the time series $Y_k$ for malicious process. $\epsilon$ models the smallest difference that the time series mean of benign processes should have from the time series mean of malicious processes. $\epsilon$ is given by the following equation:

$$\epsilon = \frac{1}{|T|} \left\{ \sum_{i=1}^{|T|} (\mu_b(Y_i) - \frac{\sum_{i=1}^{|T|} \mu_b(Y_i)}{|T|})^2 \right\}^{\frac{1}{2}}$$  \hspace{1cm} (4.7)$$

In equation 4.6, $cv_{b,m}(Y_k)$ represents the coefficient of variation of the elements of time series $Y_k$ for both benign and malicious processes. $|Y_k|$ is the value of total number of elements in the time series $Y_k$. $Z$ is given by:

$$Z = \left\{ z \mid (\mu_b(Y_k) - 3\sigma_b(Y_k)) < z < (\mu_b(Y_k) + 3\sigma_b(Y_k)) \right\}$$  \hspace{1cm} (4.8)$$

Having developed a formal definition of \textit{genetic footprint}, let us now demonstrate the usefulness of the above-mentioned two step feature selection methodology. A first look at Figures 4.4(a) and 4.4(d) might provide the temptation to use both of them in the features’ set because their time series mean is significantly different in benign and malware samples. However, once their variance is plotted in Figures 4.4(b) and 4.4(e), and their corresponding coefficient of covariance in Figures 4.4(c) and 4.4(f) respectively, two important conclusions can be drawn: (1) the parameter \textit{page_table_locks} should be made part of the \textit{genetic footprint} because of its low $cv$ both in benign and malicious processes, and (2) the parameter \textit{majflt} should not be considered for inclusion in \textit{genetic footprint} because of its relatively high $cv$ value in benign samples. To further confirm these findings about the effectiveness of selected and rejected parameters, their distributions are plotted in Figures 4.4(g) and 4.4(h). It is clear that \textit{page_table_locks} with a low $cv$ has a gaussian distribution; while \textit{majflt} with a high $cv$ does not have a well known distribution. It is known that in case of a gaussian distribution, 95.45% of the values lie between the interval $\mu \pm 3\sigma$. On the basis of this analysis, the following rule set is evolved to make a feature part of the \textit{genetic footprint}:

1. Its time series mean is significantly different for both benign and malicious processes,

2. coefficient of variation is less than 3,

3. 95.45% of its values are within the interval $\mu \pm 3\sigma$.

Once these rules are iteratively applied to the above-mentioned 36 features, 20 more features are discarded; as a result, \textit{genetic footprint} now consists of 16 features. Just to efficiently utilize the space, the plots of three other parameters are shown: (1) total\_vm is accepted because of low $cv$ in Figure 4.4(i), (2) link\_count in Figure 4.4(k) is rejected because of its similar pattern in benign and malicious processes, and (3) map\_count is accepted in Figure 4.4(j) because of its low $cv$. Finally, the \textit{genetic footprint} is created and the final features are included in Table 4.2. Now we
focus our attention towards identifying a classifier that meets three requirements: (1) low training and testing times, (2) high detection rate, and (3) low false alarm rate.

Table 4.2: Description of the fields constituting genetic footprint

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task→fpu_counter</td>
<td>Usage counter of floating point units</td>
</tr>
<tr>
<td>task→active_mm→map_count</td>
<td>No. of memory regions of a process</td>
</tr>
<tr>
<td>task→active_mm→page_table_lock</td>
<td>Needed to traverse &amp; manipulate the page table entries</td>
</tr>
<tr>
<td>task→active_mm→hiwaterrss</td>
<td>Max no. of page frames ever owned by a process</td>
</tr>
<tr>
<td>task→active_mm→hiwater_vm</td>
<td>Max no. of pages appeared in memory region of process</td>
</tr>
<tr>
<td>task→active_mm→total_vm</td>
<td>Size of process’s address space in terms of no. of pages</td>
</tr>
<tr>
<td>task→active_mm→shared_vm</td>
<td>No. of pages in shared file memory mappings of process</td>
</tr>
<tr>
<td>task→active_mm→exec_vm</td>
<td>No. of pages in executable memory mappings of process</td>
</tr>
<tr>
<td>task→active_mm→nr_ptes</td>
<td>No. of page tables of a process</td>
</tr>
<tr>
<td>task→utime</td>
<td>Tick counts of a process that is executing in user mode</td>
</tr>
<tr>
<td>task→stime</td>
<td>Tick counts of a process in the kernel mode</td>
</tr>
<tr>
<td>task→nvcsw</td>
<td>Number of volunteer context switches</td>
</tr>
<tr>
<td>task→nrvcsw</td>
<td>Stores the no. of in-volunteer context switches</td>
</tr>
<tr>
<td>task→minflt</td>
<td>Contains the minor page faults</td>
</tr>
<tr>
<td>task→alloclock.rawlock.slock</td>
<td>Used to lock memory manager, files and file system etc</td>
</tr>
<tr>
<td>task→fs→count</td>
<td>fs_struct’s usage count to indicate the restrictions</td>
</tr>
</tbody>
</table>

Table 4.3: Class noise of dataset used in experiments

<table>
<thead>
<tr>
<th></th>
<th>RBF-C</th>
<th>SVM POLY-K</th>
<th>SVM PUK-K</th>
<th>SVM RBF-K</th>
<th>J48</th>
<th>J-Rip</th>
<th>Class Noise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benign Instances (%)</td>
<td>31.87</td>
<td>34.65</td>
<td>44.70</td>
<td>57.71</td>
<td>8.52</td>
<td>6.91</td>
<td>10.2 (%)</td>
</tr>
<tr>
<td>Malware Instances (%)</td>
<td>14.87</td>
<td>10.28</td>
<td>25.65</td>
<td>27.08</td>
<td>6.91</td>
<td>6.91</td>
<td>6.91</td>
</tr>
<tr>
<td>Overall Class Noise</td>
<td>10.2 (%)</td>
<td>6.91</td>
<td>6.91</td>
<td>6.91</td>
<td>6.91</td>
<td>6.91</td>
<td>6.91</td>
</tr>
</tbody>
</table>

4.4.3 Classification

Recently, the role of a given dataset in selecting an appropriate classifier is being investigated [Tanwani et al., 2009] [Tanwani and Farooq, 2009]. A well known measure – class noise [Brodley and Friedl, 1999] – is defined to quantify the challenging nature of a dataset. In [Zhu and Wu, 2004], the authors have shown that the classification accuracy is more dependent on class noise instead of attribute noise. The authors classify a given dataset by using a number of well known trained classifiers and the intersection of correctly classified instances constitutes non-noisy dataset. Consequently, the intersection of all misclassified instances by all classifiers is defined as the class noise. We follow the same approach: use all instances of benign and malware processes and classify them with neural networks (RBF-Network), Support Vector Machines (SVM) with multiple kernels (Polynomial kernel, universal Pearson VII function based kernel (Puk) and Radial Basis Function based kernel (RBF)), decision tree (J48) and a propositional rule learner (J-Rip) in Wakaiko Environment for Knowledge Acquisition (WEKA) [Witten and Frank, 2002]. The results of misclassified instances are reported in Table 4.3 for all classifiers. It is obvious from this empirical study that the intersection of misclassified instances of benign and malicious processes is 3.29% and 6.91% respectively. (This leads to an overall class noise of 10.2 %). In comparison, SVM and neural networks are unable to cope with the complexity of our time series dataset. In [Witten and Frank, 2005], it is shown that J48 is resilient to class noise because it avoids over fitting during learning and
also prunes a decision tree for an optimum performance. Similarly, J-Rip also applies pruning during the formulation of rules to achieve better accuracy. Therefore, we have short listed J-48 and J-Rip.

Now we cross validate our preliminary decision of selecting J-48 and J-Rip by analyzing the classification potential of genetic footprint. (Higher detection accuracy means that genetic footprint has higher classification potential). In order to do this, renowned statistical measures (commonly used in data mining and knowledge extraction), Information Gain and Information Gain Ratio, are applied. Information gain IG for a parameter $p$ is given by the following equation:

$$IG(T_s, g) = H(T_s) - \sum_{v \in values(g)} \frac{|\{s \in T_s | V(s, g) = v\}|}{|T_s|} \times H(\{s \in T_s | V(s, g) = v\})$$

(4.9)

$T_s$ is the training samples that is used. $V(s, g)$ defines the value of sample $s \in T_s$ for parameter $g \in G$ where $G$ represents the genetic footprint that consists of 16 selected features. $|T_s|$ is the number of elements in the set $T_s$. $H$ defines the entropy of $T_s$ and is calculated using the following equation:

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log_b p(x_i),$$

(4.10)

where $b$ is the base of the logarithm and Euler’s number $e$ is used as base of the logarithm. $X$ is a discrete random variable, $p$ denotes the probability mass function of $X$, and $n$ is the total number of values that $X$ can have.

The information gain of genetic footprint is plotted in Figure 4.5(a). It is seen that information gain of most of the features is relatively high. It is a well known fact in data mining research: decision tree based algorithms and propositional rule learners achieve high classification accuracy for classification of instances where parameters have reasonably high information gain [Witten and Frank, 2005]. However, their accuracy can significantly degrade, even if the selected features have high information gain because they might have large number of distinct values. (In this case, the features of genetic footprint can also possibly have large number of distinct values). In this scenario, an effective decision tree is not created because of relatively high bias towards features that can have large number of distinct values. In order
4.5 Experiments and Results

To remove this bias, another measure – information gain ratio (IGR) (defined in the following) – is used.

\[
IGR(Ts, g) = \frac{IG(Ts, g)}{-\sum_{s} \frac{|s \in Ts|}{|Ts|} \cdot \log_b \left( \frac{|s \in Ts|}{|Ts|} \right)}
\]  

(4.11)

If information gain ratio is approximately 1, decision trees and propositional rule learners give good classification accuracy. IGR of features in the genetic footprint is plotted in Figure 4.5. It can be seen that IGR of most of the features is near to 1, and hence it concurs with our preliminary decision to use decision tree algorithms and propositional rule learning algorithms.

4.5 Experiments and Results

In this section, the accuracy of two short listed classifiers is evaluated using the features in genetic footprint. Specifically, following issues are discussed chronologically: (1) the overall accuracy of a classifier using genetic footprint, (2) the impact of increasing detection time on the classification accuracy, (3) detection of a malicious process if it mostly behaves like a benign process and performs malicious activity for a short duration later during its execution, (4) the false alarm rate of the proposed system, (5) detection of the malicious processes before their exit even if they finish within 30 ms of their launch, (6) the processing overheads of logging, training and testing of the proposed system, and (7) a comparison of the accuracy of the proposed scheme with other sequence calls based solutions. Later in Section 4.6, the “robustness” of the proposed system is discussed to evasion attempts if a malicious process tries to imitate the behavior of a benign process.

In order to evaluate the effectiveness of genetic footprint in detecting malware using J48 and J-Rip, a stratified 10-fold cross validation strategy is used. The dataset is divided into 10 folds – a fold on the average consists of 11 malware and 10 – 11 benign samples – and classifiers are trained on 9 folds and are tested on the remaining 1 fold. WEKA [Witten and Frank, 2002] is used for this 10 fold analysis to remove any bias in evaluation due to implementation of a classifier.

It is important to note that most of the processes finish before 15 seconds, hence it is important to use the concept of step size to ensure that learning is not biased towards the processes running longer. Therefore, the step size is defined as:

\[
step\_size = \frac{x}{n}
\]  

(4.12)

where step_size is the number of samples after which a sample of a process is selected for training, x is total number of samples of a process and n is number of selected samples. For our experiments, we have taken n=30. Step size helps us to select training samples at almost equally spaced intervals.

In a typical two-class problem, such as malicious process detection, the classification decision of an algorithm may fall into one of the following four categories: (1) true positive (TP), classification of a malicious process as malicious, (2) true negative (TN), classification of a benign process as benign, (3) false positive (FP), classification of a benign process as malicious, and (4) false negative (FN), classification of a malicious process as benign.
The detection accuracy of proposed system is reported using three separate metrics: (1) detection rate (DR), (2) false alarm rate (FAR), and (3) detection accuracy (DA). These metrics are defined in the following:

\[ DR = \frac{TP}{TP + FN} \]  

\[ FAR = \frac{FP}{FP + TN} \]  

\[ DA = \frac{TP + TN}{TP + TN + FP + FN} \]

We now discuss the issues raised in the beginning of this section.

4.5.1 Overall classification accuracy using genetic footprint

The accuracy of the two classifiers is reported – using genetic footprint – in Table 4.4\(^1\). The average results and the results for testing each fold are reported. In Table 4.4, the results are tabulated for window sizes of 10, 30, and 100 respectively. The “window size” defines the number of instances considered from each testing sample before making a decision. In case of 10, a classifier just takes first 10 instances of each feature in the genetic footprint and makes a decision. Remember, a process is declared as malicious or benign on the basis of the majority vote within a given window. In a window of 10, if six classifications are malware, the process is declared as malware. In case of equal votes for malware and benign, the system defers the decision to the next window. It can be seen that J-Rip – in case of a window size of 100 – DR, FAR and DA are 93.7%, 0% and 96.65% respectively. (For J48 these parameters are 1-4% inferior). To the best of our knowledge, no existing dynamic analysis system has achieved such detection and false alarm rates. In order to get a better insight, we plot ROC curves of different classifiers in Figure 4.6 for a window size of 100. (Note that a fold number in Figure 4.6 corresponds to the same fold number in Table 4.4.) The detection rate of J-Rip of 10 folds varies from 0.79 to 1 for a false alarm rate of 0. In comparison, on some folds, J48 has a false alarm rate of 0.1 to 0.2.

In order to do a comparative study about the effectiveness of other classifiers, we plot ROC of commonly used classifiers – RBF neural networks [Fisch et al., 2010] and Naive Bayes [Menahem et al., 2009] – for malware detection in Figures 4.6(c) and 4.6(d) respectively. It is obvious from the figure that both of them are unable to provide accurate detection with low false alarm on all folds on our genetic footprint dataset.

4.5.2 The impact of increasing detection time on the classification accuracy

It is evident in Table 4.4 that as the detection time increases from 10 to 30, FAR is significantly reduced and DR also improves – resulting in an overall improvement.
Table 4.4: Accuracy results of J48 and J-Rip on each of the 10 folds

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Time (ms)</th>
<th>J48</th>
<th>J-Rip</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fold 1</td>
<td>Fold 2</td>
<td>Fold 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TP</td>
<td>10</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>FP</td>
<td>00</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>TN</td>
<td>10</td>
<td>09</td>
<td>09</td>
</tr>
<tr>
<td>FN</td>
<td>01</td>
<td>01</td>
<td>00</td>
</tr>
<tr>
<td>DR (%)</td>
<td>91.0</td>
<td>91.0</td>
<td>100</td>
</tr>
<tr>
<td>FAR (%)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>DA (%)</td>
<td>95.2</td>
<td>90.5</td>
<td>95.2</td>
</tr>
<tr>
<td>TN</td>
<td>10</td>
<td>09</td>
<td>09</td>
</tr>
<tr>
<td>FP</td>
<td>00</td>
<td>01</td>
<td>00</td>
</tr>
<tr>
<td>TN</td>
<td>10</td>
<td>09</td>
<td>09</td>
</tr>
<tr>
<td>FN</td>
<td>00</td>
<td>01</td>
<td>00</td>
</tr>
<tr>
<td>DR (%)</td>
<td>91.0</td>
<td>91.0</td>
<td>100</td>
</tr>
<tr>
<td>FAR (%)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>DA (%)</td>
<td>90.5</td>
<td>90.5</td>
<td>95.2</td>
</tr>
<tr>
<td>TN</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>FP</td>
<td>00</td>
<td>01</td>
<td>00</td>
</tr>
<tr>
<td>TN</td>
<td>10</td>
<td>09</td>
<td>09</td>
</tr>
<tr>
<td>FN</td>
<td>00</td>
<td>01</td>
<td>00</td>
</tr>
<tr>
<td>DR (%)</td>
<td>91.0</td>
<td>91.0</td>
<td>100</td>
</tr>
<tr>
<td>FAR (%)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>DA (%)</td>
<td>90.5</td>
<td>90.5</td>
<td>95.2</td>
</tr>
<tr>
<td>TN</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>FP</td>
<td>00</td>
<td>01</td>
<td>00</td>
</tr>
<tr>
<td>TN</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>FN</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>DR (%)</td>
<td>91.0</td>
<td>91.0</td>
<td>91.0</td>
</tr>
<tr>
<td>FAR (%)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>DA (%)</td>
<td>95.2</td>
<td>95.2</td>
<td>95.2</td>
</tr>
<tr>
<td>TN</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>FP</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>TN</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>FN</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>DR (%)</td>
<td>91.0</td>
<td>91.0</td>
<td>91.0</td>
</tr>
<tr>
<td>FAR (%)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>DA (%)</td>
<td>95.2</td>
<td>95.2</td>
<td>95.2</td>
</tr>
<tr>
<td>TN</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>FP</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>TN</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>FN</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>DR (%)</td>
<td>91.0</td>
<td>91.0</td>
<td>91.0</td>
</tr>
<tr>
<td>FAR (%)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>DA (%)</td>
<td>95.2</td>
<td>95.2</td>
<td>95.2</td>
</tr>
</tbody>
</table>

4.5 Experiments and Results
of 1-2% in detection accuracy. Moreover, if the detection time is allowed to increase beyond 30 to 100, a significant improvement in the FAR is observed.

This behavior of proposed system can be easily explained if the basic motivation of a malware writer is understood: do the intended damage as quickly as possible. In the selected dataset about 15% malware finish in less than 100ms; as a result, most of them start malicious activity just after their launch; as a result, they can be detected within 30 ms.

4.5.3 Detection of backdoors-type processes

Let’s now consider the typical behavior of backdoors – they might start their first malicious activity after 1000 ms. Recall that the proposed technique do not just perform a one time check at the start of a process; rather, it keeps on sliding the “window” to continuously invoke the classifier. As a result, system will detect an anomaly within 30 ms of its launch (assuming a window size of 30 ms). This phenomenon can be seen in Figure 4.4(l) that plots parameter exec_vm of a Linux backdoor Excedoor. It is clear that the process behaves like a benign one till 3148
ms and then it starts its malicious activity. The proposed system has detected it as a malware. To conclude, proposed system can detect a malicious activity because of its ability to do continuous real-time monitoring.

4.5.4 False alarm rate of proposed system

It is depicted in Table 4.4 that for a window size of 100 ms, the FAR of J48 and J-Rip is 4% and 0% respectively – 2-6% reduction if a window size of 10 ms is chosen. This shows that 10 ms is relatively a small window size and within this time interval the true behavior of a process can’t be learnt. A low false alarm is very important from the usability perspective; otherwise, the users will be annoyed if their legitimate applications are frequently stopped.

4.5.5 Detection of tiny malicious processes

It is already mentioned that 17 (15%) of malware finish in less than 100 ms, 11 malware finish in less than 30 ms and 3 even in less than 10 ms. Now it becomes a real challenge to detect them while they are still executing. A separate set of experiments are conducted to investigate the facts: how many of 17 malware that finish in less than 100 ms are correctly classified. The proposed technique is able to successfully classify all 14 malware processes that execute for more than 10 ms. In comparison, 1 out of 3 malware that finish before 10 ms are misclassified. (Remember it is a tight constrain on the system to give a decision before termination of a malware). As a result, DR for this challenging dataset is 93.7% and this substantiates the claim that it can accurately classify even those malware samples that terminate very quickly.

4.5.6 Processing overheads of proposed scheme

In an online run-time system, it is very important to measure the processing overhead of each module of the proposed system. The processing overhead of the system is a sum of feature logging and testing times. Training time is not critical because it happens only once at the beginning and then after every one hour. The feature logging time is 40 microseconds for each instance (it includes the context switching time of 6 microseconds). J48 and J-Rip take 18 ms and 30 ms during the training phase respectively. Moreover, J48 and J-Rip have 45 and 100 microseconds testing time respectively. If the features’ logging time is added to the testing time, then J48 and J-Rip take 85 and 140 microseconds per instance. The results are intriguing enough to make serious efforts to embed the detection module inside the Linux kernel for online analysis and detection.

4.5.7 A comparison of the accuracy of the proposed scheme with other sequence calls based solutions

The focus of this study is to compare the detection accuracy of our framework with recently proposed sequence calls based malware detection techniques. Most of these techniques use n-gram (typically a sequence of 5 to 10 system calls) based representation for dynamic malware detection [Qian and Xin, 2007], [Gao et al., 2006] [Forrest et al., 1996], IMAD [Mehdi et al., 2009], and Hyper-grams [Mehdi
et al., 2010]. The techniques selected for the comparison are: n-grams (ranging from 5 to 10), IMAD and Hyper-grams. J48, J-Rip and Baysianet are used for classification of n-gram based sequence representation. These techniques classify a process by matching its system call sequences to that of benign or malware call sequences.

IMAD is a realtime and efficient in-execution malware detection scheme. It uses n-gram representation and optimizes the learning process (using a genetic algorithm) for system call sequences that are present in both benign and malicious processes. It tunes different parameters to improve the accuracy of classification.

Hyper-grams technique uses variable length n-grams (called Hyper-grams) instead of using fixed length n-grams. It is a generalized scheme in which n-gram of the trace of a process is visualized in a k-dimensional hyperspace by following the sequence of its system calls. Here, k represents the number of unique system call sequences followed by a process. The path taken by a process in hyperspace is used to model its behavior by matching it with the paths of benign and malicious processes.

In order to evaluate these techniques, a log of system calls of all benign and malicious processes is collected by executing them on Linux OS. The malware detection accuracy (DA) and false alarm rate (FAR) of these schemes are reported in Table 4.5. One can see that n-gram with J48 achieves on the average DA and FAR of 80.13% and 5.63% respectively. (For J-Rip DA is 1% inferior but FAR is 2% superior). The maximum accuracy (86%) with a 0% FAR is achieved by Hyper-grams technique. In comparison, our technique achieves more than 96% DA with a 0% FAR.

### 4.6 Evasion

The evasion of this system is also studied from two different perspectives: (1) the robustness of proposed scheme to evasion attempts if a malicious process tries to imitate the benign behavior for various parameters, and (2) access restrictions to task_struct parameters so that a process is unable to access or modify their values directly. The answer to the second question is critical because a malware has to first estimate the benign genetic footprint before imitating it. (Note all features of genetic footprint are system and machine dependent and hence must be learned for each system).
4.6 Evasion

4.6.1 Robustness of proposed scheme to evasion attempts

In this section, it is analyzed that how robust the proposed scheme is to evasion attempts if a malicious process tries to imitate the benign behavior for various parameters. In order to systematically undertake the study of robustness of the set of 16 features of genetic footprint, different features in the genetic footprint are replaced of a malware with that of corresponding features in the genetic footprint of a benign file. This is the logical way of imitating a benign process as malicious. The effect of this “forgery” on the accuracy of proposed system is analyzed. (All results are reported for a window size of 100). It is observed that once 4 features are forged, only 3% more malicious processes are misclassified. If 6 features are forged, accuracy further deteriorates by 2% and a total of 11% malware processes are misclassified as benign. Finally, the accuracy becomes totally unacceptable once 8 features are forged. The accuracy deteriorates by 13% (becomes 83%), which is still very competitive compared with existing state-of-the-art run-time systems.

Here, It should be emphasized that the above-mentioned forgery is done because the values of benign and malware genetic footprints are known. Remember that most of these features depend on a particular configuration of a system like cache, RAM, secondary storage, processor, paging mechanism, stack and heap managers etc and therefore, they must be estimated for each host. These values can change from one host to another. As a result, a crafty attacker has to first estimate these values for a particular system on which it wants to imitate benign process. This demands that malware be equipped with hooks to log different fields of genetic footprint (task struct) for benign processes. This brings us to the second issue related to access restrictions and it is discussed in the following.

4.6.2 Access restrictions to task_struct parameters

The most dangerous threat to the security of a kernel and its structures is posed by kernel rootkits. These rootkits can hide themselves by subverting “reference monitor” of an operating system [Castro et al., 2006]. As a result, they tamper with OS functionalities to launch various types of well known attacks: OS backdoors, stealing private information, escalating privileges of malicious processes, and disabling defense mechanisms. Such rootkits can surely get an access to the fields of task_struct, and then enable a running malware processes to learn genetic footprint of benign processes and then it can directly modify its genetic footprint to evade the system.

But the above-mentioned problem is well known in the OS security community for years now. Recently, researchers have proposed a number of novel and effective methodologies that can stop rootkits and other malicious process that try to access or modify the kernel structures. Some of these schemes are: kernel module signing [Microsoft, 2007], W⊕X [PaX, 2010], NICKLE [Riley et al., 2008], and SecVisor [Seshadri et al., 2007]. Their authors have shown that these schemes effectively and efficiently protect kernel structure from illegitimate access and modifications in all possible scenarios. Hund et al. in a recent paper [Hund et al., 2009] have introduced the concept of return oriented rootkit, which circumvents all above-mentioned schemes of protection of kernel structures. Luckily, Wang et al. have proposed a solution in [Wang et al., 2009] that again blocks the access of these return oriented rootkits to the kernel structures.
To conclude, a number of schemes exist that can stop the malicious codes to access kernel structures. As a consequence, these schemes make it not-so-easy for malware writers to learn \textit{genetic footprint} of a benign process on a given system. As a result, it becomes a challenge for a “crafty attacker” to forge the features and evade the proposed scheme. Nevertheless, this fact is acknowledged that rootkits are (by any means) not a solved problem and in future they can pose a potential threat to this proposed scheme.

4.7 Related Work

Today, most of the commercially available antivirus programs identify a malware on the basis of strings or instruction sequences that are typical to that particular malware [Szor, 2005]. These strings or instruction sequences define the signature of the malware file. As this technique is based upon syntactic appearance of a malware, so it is easily evaded by code obfuscation and polymorphism [Christodorescu and Jha, 2004] [Szor, 2005]. To overcome this short coming of signature based detection schemes, researchers proposed some higher order properties that can capture the intrinsic characteristics of a program and are thus difficult to disguise. One of the most commonly used such property is the n-grams character distribution of a file [Li et al., 2007] [Li et al., 2005] [Shafiq et al., 2008]. This property is very useful in identifying embedded malware in benign files. Another technique to detect polymorphic variants of a malware is control flow graph [Bruschi et al., 2006] [Kruegel et al., 2006]. More sophisticated static analysis approaches utilize code templates that capture the malicious functionality of certain malware families. For this purpose symbolic execution [Kruegel et al., 2004], model checking [Kinder et al., 2005], or compiler verification [Christodorescu et al., 2005] and semantic aware [Preda et al., 2008] techniques are applied to recognize arbitrary code fragments that implement a specific function. These techniques learn a functionality independent of the specific machine instructions that express it.

Although a number of sophisticated static analysis techniques have been developed that show promising results but they also face some significant shortcomings. For example, as the malware programs rely heavily on run-time packing and self-modification of code, the instruction present in the binary on a disk are different than those executed at runtime. Although generic unpackers [Royal et al., 2006] can be helpful in obtaining the actual instructions, binary analysis of obfuscated code is still very difficult [Moser et al., 2007]. Furthermore, many advanced static analysis approaches are extremely slow (at times of the order of minutes [Christodorescu et al., 2005]), and are thus unsuitable for detection in real-world deployment scenarios.

Dynamic analysis techniques intend to detect malicious processes by analyzing the execution of a program or the effects that this program has on the operating system. The former generally utilizes the system calls sequence of a program to analyze its behavior. The seminal work reported in [Forrest et al., 1996] leveraged the temporal pattern of system calls to discriminate a malicious process from a benign one. Later, a number of enhanced variants of the above-mentioned technique have been proposed in [Wang et al., 2007], [Casas-Garriga, Diaz, and Balcazar, Casas-Garriga et al.] and [Mutz et al., 2006]. Another technique is the code-based static analysis of system calls proposed in [Wagner and Dean, 2001]. In [Giffin et al.,
The authors have used spatial information in the arguments of parameters to identify malicious processes. Yet another technique uses system calls stack and program counter information [Feng et al., 2004] [Giffin et al., 2004]. An example of the technique that uses the effects of a running program on the operating system is Strider GhostBuster [Wang et al., 2005]. It has the ability to detect certain kinds of rootkits that hide themselves from the user by filtering the results of system calls. It does so by comparing the view of the system provided by a possibly compromised OS to the view that is gathered when accessing the file system directly. A novel worm detection scheme by utilizing neural networks is proposed in [Stopel et al., 2009]. The authors have used performance counters\footnote{The performance counters of operating system are also used by researchers for systems’ bottlenecks detection [Ahmed et al., 2010]} of an operating system to train the neural network to detect the unknown worms in realtime. Another dynamic malware detection technique is based on the analysis of disk access patterns [Paul et al., 2005] as the malicious processes usually access disk in a manner that can be distinguished from a benign program. The biggest advantage of this approach is that it can be incorporated into the disk controller, and is thus difficult to bypass; however, it can detect a malware only if it modifies a large numbers of files stored on the disk. In a recent work [Park et al., 2010], the authors build clusters of transaction audit data streams on hosts to detect any malicious activity.

Such approaches show promising results but their large processing overheads and detection time (sometimes in the order of even minutes) make them infeasible for real world deployment. Moreover, these techniques are also evadable. A malware can easily change its sequence of system calls or add irrelevant calls and these schemes fail in such situations. Similarly, return oriented rootkits can evade the systems like Strider GhostBuster. In comparison, our technique uses a novel concept of \textit{genetic footprint} that enables it to detect a malware in less than 100 ms. Moreover, it provides more than 96\% accuracy with 0\% FAR.

4.8 Conclusion and Future Work

In this chapter, a novel concept of \textit{genetic footprint} of a process is used to detect malicious processes at run time. We have formally defined the \textit{genetic footprint} and provided a comprehensive analysis of the selection process of the parameters that constitute it. As a result, 16 task structure parameters have been selected from 118 parameters after a thorough statistical analysis. A suitable classifier is selected on the basis of the complexity of dataset and information gain analysis of \textit{genetic footprint}. The idea of using \textit{genetic footprint} for detection of malware is validated on a dataset of 105 benign processes and 114 recently collected malware processes for Linux. The results of experiments demonstrate that the proposed system achieves a detection accuracy of 96\% with 0\% false alarm rate. Moreover, it detects a malicious process in less than 100 milliseconds of the start of a malicious activity. To the best of our knowledge, this is the shortest detection time achieved to date by any dynamic detection scheme. The presented technique is robust to runtime evasion attempts and has small processing overheads. In future, it has been planed to embed the scheme in the kernel of Linux and evaluate its effectiveness for other operating systems as well.
Table 4.6: Malware dataset with categories, names and file sizes [VX-Heavens, 2009] [Offensive-Computing, 2009]
Chapter 5

A Hybrid Framework for Malware Detection on Smartphones using ELF Structural & PCB Runtime Traces

5.1 Introduction

Sophisticated mobile computing devices – tablets and smartphones – are escalating their processing and storage capabilities and gaining popularity simultaneously in the consumer market as well as in the mysterious world of hackers and malware writers. Existing commercial anti-malware products and non-signature-based techniques are unable to detect zero-day and polymorphic malware with high accuracy, low false-alarm-rate and small detection-overheads on smartphones. To this end, we present a novel hybrid framework on smartphones which uses structural traces of executables and PCB traces of executing processes in OS kernel for detecting zero-day and polymorphic malware. This solution is presented on Linux based smartphone OpenMoko but it is directly portable to Ubuntu based superphones presented in 2013.

Mobile devices and smartphones have gained overwhelming popularity in recent years. The analysts expect that there shall be 5 billion devices in the market till 2015 [Liu et al., 2009]. With this boom of smartphones, malicious softwares are gradually becoming significant threats. For instance, according to the latest survey of Mocana Corporation – Mobile & Smart Device Security Survey 2011 [Vamosi and Stammberger, 2011] – 74% of the respondents cited the viruses – as the most widespread vector of attack on their smartphones. The data leakage and Trojans were cited as the second and third level security threats by 59% and 55% correspondents respectively. In security report of Symantec Corporation published in 2010 [Fossi, 2010], 45% more vulnerabilities and hundreds of new malwares are reported for smartphones and other mobile devices.

To counter this threat, world’s renowned antivirus / anti-malware manufacturing companies e.g. Kaspersky, F-Secure, BullGuard, AVG, Lookout etc. have launched their security software products for hand-held mobile devices [Smartphone-antivirus, 2011]. But all these anti-malware softwares are signature-based and hence are un-
able to detect zero-day malware [Sukwong et al., 2010]. Traditionally, it has been observed in computers that majority of new malicious softwares are repacked and polymorphic version of some existing malwares [Taha, 2007]. Furthermore, they can successfully evade existing commercial-off-the-shelf antivirus (COTS-AV) using polymorphism and code obfuscation techniques.

Mobile devices have well-known limitations and security challenges as compared to traditional desktop computer systems e.g. limited computational power, less memory and low-power battery [Oberheide and Jahanian, 2010]. In this context, a viable security system for smartphones must incorporate zero-day (previously unknown), repacked and polymorphic malware detection with high detection accuracy and low false-alarm-rate. Furthermore, it must be light weight and resilient to evasion attempts. To this end, we propose a hybrid framework that incorporates both static and dynamic malware analysis and detection techniques. The framework consists of two components: EST and PRT. EST extracts the static, structural traces of executables to devise a non-signature-based zero-day (previously unknown) malware detection scheme for ARM\(^1\) Linux malware\(^2\) on smartphones. In contrast, PRT detects malicious processes at runtime, most of which are encrypted, repacked and polymorphic variants of previously known malwares. The PCB runtime trace, used by PRT, consists of selected parameters – maintained inside the process control blocks of the kernel of an OS for each running process – that define the semantics and behavior of an executing process.

Linux is getting increasingly popular in the world of smartphones as well – Google’s Android is a typical example. Therefore, we present our framework on Linux based smartphone – OpenMoko Free Runner – running Angstrom distribution on ARM architecture. Due to the scarce availability of malware, we use cross-compiled x86-Linux malware for ARM architecture for the empirical evaluation of our framework. These cross-compiled malwares are sufficient to provide a proof-of-concept for zero-day malware detection scheme i.e. EST. But, our framework is also elixir for nuisance caused by polymorphic variants of malware. Therefore, we use a customized malware evolution technique [Noreen et al., 2009] to generate hundreds of polymorphic variants of malware for ARM-Linux platform.

We evaluate both components of our hybrid framework separately. For evaluation of EST, we perform a forensic analysis of cross-compiled malwares by using information theoretic measure – KL divergence – to substantiate the classification potential of ELF structural trace features. We remove redundant features by using pre-processing filters and subsequently use Discrete Wavelet Transformation for feature selection. Our experiments, which employ multiple machine-learning-classifiers, show that EST achieves 99% accuracy with 0% false-alarm-rate on both singleton and existere malware datasets. To prove the thesis of in-execution malware detection through PRT, we conduct another forensic study to shortlist 14 – out of a total of 118 – task structure parameters that can distinguish between the execution traces of benign and malware processes. The empirical results show that, using renowned machine learning classifiers, PRT achieves 96% detection-accuracy with 0% false-alarm-rate within 1 sec of process’s execution. Last but not the least, our technique utilizes partial knowledge available at a given time during process execution. There-

\[^1\text{ARM: Advanced RISC Machine (An energy efficient processor architecture being used for smartphones)}

\[^2\text{Cross compiled Linux x86 malware for ARM architecture}\]
fore, an OS kernel can also devise mitigation strategies. In the end, we show that the components of our framework are also robust to evasion attempts.

5.1.1 Organization of the chapter

Rest of the chapter is organized as follows. We present a brief description of related work in Section 5.2. We provide details of ARM malware evolution and dataset preparation in Section 5.3. Section 5.4 explains the architecture of our hybrid framework. We describe EST as a case study in Section 5.5 and its classification results in Section 5.6. We discuss PRT and its classification results in Section 5.7 and 5.8 respectively. We conclude in Section 5.9 with a preview of our future work.

5.2 Related Work

To overcome the shortcomings of signature-based schemes, researchers have done significant work for identification & typecasting of malware, their taint-analysis and detection on multiple smartphones’ platforms [Yan et al., 2009][Giffin, 2010] e.g. Apple’s iPhone [Damopoulos et al., 2011], Google’s Android [Davi et al., 2011][Felt et al., 2011a] [Shabtai et al., 2010a] and Symbian OS [Schmidt and Albayrak, 2008]. These techniques can be segregated into three broader categories: (1) static, (2) dynamic and (3) behavior-based analysis & detection schemes.

Most of the static analysis based techniques utilize information embedded in a given file or code templates to capture the functionality of a specific malware family. In [Schmidt et al., 2009a], the authors disassembled the executables on Android, extracted the system and library function calls from benign files and compared them with the relative function call list of malicious files using machine-learning-classifiers. They share malicious features’ vector among multiple mobile hosts for collaboration and detection of similar malicious files on other hosts. This scheme provides 96% detection-accuracy with 10% false-positives. The authors of [Schmidt et al., 2009a] extended their function-calls-based solution to Symbian OS in [Schmidt et al., 2009b]. In this scheme, they employed their clustering algorithm, Centroid, to identify benign and malicious executables. They achieved 70-90% detection-accuracy with 0-20% false-alarm-rate.

The authors of [Blasing et al., 2010] have presented a framework, Android Application SandBox, for malicious software detection on Android. It consists of two phases. In the first phase (at mobile host), it performs static analysis of user applications, converts their class files into java source code, searches for malicious patterns and marks them as benign or malicious. In the second phase, applications (APK files) are sent to a remote server in compressed format. The server then decompresses and executes them in the sandbox, logs their low-level interactions with OS and analyzes the logs for malicious patterns by employing the clustering algorithm. However, in case of packed binaries, these techniques fail because packed-file on a disk are different from their unpacked executable counterparts [Taha, 2007]. Secondly, the techniques are extremely slow and computationally expensive due to files disassembly and network communication which makes them unsuitable for real-time deployment on smartphones [Schmidt et al., 2009a].

Dynamic malware detection techniques intend to detect malicious programs by monitoring their execution patterns / performing the taint-analysis and estimating
their impacts on the OS. As a result, they are resilient to code obfuscation or polymorphism techniques. In general, two types of dynamic techniques have been proposed: (1) system-call-sequencing and (2) flow graphs. The former techniques generally build behavioral models on the basis of the sequence of invoked system-calls and the information contained in their parameters [Ahmed et al., 2009] [Yin et al., 2007] [Mehdi et al., 2010] and [Li et al., 2008]. On smartphones, Paranoid Android (PA) [Portokalidis et al., 2010] is the best example of API based malware detection. PA uses the concept of decoupled security for smartphones. It records the system-calls, compresses and transmits them to a remote server in cloud for further processing and classification.

The basic philosophy of flow graphs is to build taint graph of a running program by organizing its data flow model [Yin et al., 2007][Kirda et al., 2006]. This technique is now commonly used to detect polymorphic variants of malware on smartphones and x86 systems. In [Grace et al., 2010], the authors have proposed a system-calls-based framework, WoodPecker, on Android that employs control flow graph to detect capability leaks in different procedures and applications in a systematic fashion. In [Enck et al., 2010], the author have proposed a tool TaintDroid for smartphones which performs taint-analysis and identifies misbehaving and user data stealing applications using flow graphs. Although, dynamic malware detection approaches show promising results but they have the following shortcomings: (1) large processing overheads and malware detection time [Kolbitsch et al., 2009], (2) vulnerability to mimicry attacks (adding irrelevant and factitious system-calls to fail these schemes) [Parampalli et al., 2008] and (3) in the case of constrained mobile environment, decoupled security solutions increase processing overhead on handsets (logging, compressions etc.) that leads to network overloading.

Smartphones security researchers have also proposed some behavioral approaches to detect malware. SmartSirm [Cheng et al., 2007] is a collaborative virus detection and alert system for smart-phones. It collects the information of communication activity from smartphones and performs joint analysis to detect both single-device and system-wide abnormal behaviors. In [Miettinen and Halonen, 2006], the authors have proposed a host-based intrusion detection techniques using the events of OS (i.e. CPU activity, memory consumption etc.), application level events (e.g. user actions etc.) and communication level events (e.g. sending and receiving of file and messages etc.). The authors of [Schmidt et al., 2009] have proposed an anomaly based detection framework on Symbian which is based upon OS events e.g. free RAM, user inactivity, process count, CPU usage and sent sms count etc. The system proposed in [Di Cerbo et al., 2011] detects malicious applications on Android by checking and monitoring its set of exposed permissions. In [Damopoulos et al., 2012], [Shabtai et al., 2012] and [Shabtai and Elovici, 2010], the authors have proposed some other behavioral based malware detection schemes. The schemes are very expensive and resource-exhaustive. They consume significant resources for events logging, processing and detection of misuse and malicious executables.

Now we focus our attention to the malware dataset preparation that has been used to test our proposed hybrid framework.
5.3 Polymorphic Malware Dataset

Due to the unavailability of malware for ARM Linux based mobile devices – OpenMoko toolchain is used to cross-compile the x86 Linux malware for ARM architecture. To detect packed and polymorphic variants of malware, the proposed framework uses a customized malware evolution technique to generate multiple distinct malware signatures. The definitions of distinct and polymorphic malware signatures are given below.

**Malware signature definition.** The binary pattern of the machine code or an algorithm or hash (a number derived from a text or binary source) that uniquely identifies a particular malware [PCMag., 2012][Landesman, 2012].

**Polymorphic malware definition.** The malware or malicious code that changes its attributes and instructions in such a way that it becomes undetectable by signature and behavior-based anti-malware and intrusion detection systems [Radcliffe, 2007].

It can be deduced from the above definitions, that if the source code of a specific malware is evolved in such a way that the modification doesn’t effect its logical and functional flow, then after every modification and compilation, a new malware variant or signature is produced [Noreen et al., 2009]. Using malware evolution, numerous variants of ARM Linux malware are prepared and used to validate the static and dynamic malware detection techniques presented in this chapter.

5.3.1 Malware Evolution: Silvio’s Case Study

For the generation of polymorphic malware dataset, we have presented the process flow of malware evolution scheme in Figure 5.1. This technique evolves new malware variant from the source code of already existing malware. The evolution process of Silvio virus is discussed here in detail. The work-flow of malware evolution process is given stepwise.

![Figure 5.1: Block diagram of malware evolution process](image-url)

1. **Abstract Representation.** The abstract representation of a selected malware – is the initial step. This high-level abstract presentation demands an in-depth understanding of the functional specification, structure and flows of the source code of malware, to maintain the quality and precision of evolved malware variants. In this step, all variables, constants and code instructions are identified (and inserted) that probably can be modified in different variants of a specific malware.
2. Generation of templates with random variables. In Algorithm 5.1, pseudo-code of Silvio malware is given for abstract representation that contains two variables \( x \) and \( y \) and different code instructions e.g. `sleep`, `malloc`, `sprintf` etc. which use these variables. The source code of Silvio is compiled including these newly added statement and a template binary executable of Silvio virus is generated.

```
Algorithm 5.1 Pseudo-code of evolved malware program template

main()
{
    x ← 1, y ← 0
    while true do
        x ← x + rand(), y ← y + rand()
        if x > y then
            char * Name ← ”Silvio”
            sprintf(Malname,”%s%d”, Name, x)
            char * fictM ← (char*)malloc(x)
            sleep((rand()%5 + 1))
            ..........
            Other instructions of the malware source code
            free(fictM)
        end if
    end while
}
```

3. Selection of appropriate combination of random values. In this step, randomization, recombination, selection and validation of different values of variables are discussed – in [Noreen et al., 2009] authors have used genetic algorithms for this purpose. All possible and valid combinations of variables’ values are generated and evaluated because similar and invalid values of a variable may generate invalid or similar malware signatures.

4. Editing of data section to insert values for variables. After predicting a valid combination, the validated values of variables are converted back to machine-level code. Multiple copies of template binary executable of Silvio are prepared and data section of every variant file is edited to include the suitable combination of \( x \) and \( y \) (sequentially chosen from the list of predicted values).

5. Testing of generated variants. We execute these variants of Silvio on OpenMoko smartphone and test the validity of malware. Testing the malware files by execution on OpenMoko is a tedious procedure because it requires reinstallation of fresh OS for another malware variant’s testing.

After the completion of Silvio’s evolution case study, we briefly present the descriptions of all evolved polymorphic malware.

### 5.3.2 Evolved Malware for ARM-V4t Platform

Using the above malware evolution scheme, we have evolved the variants of following malware.
**5.3 Polymorphic Malware Dataset**

**Dataseg.** It is a parasitic virus that infects data segment of host file, the section headers and section offsets. The malware is based on the infection techniques proposed by Silvio Cesare.

**I’MSick.** This virus writes its code to the victim file and actual code of the victim file is followed by the virus code. Every infected file itself becomes an injector, which keeps on infecting the vulnerable files.

**Kaiowas10 & Kaiowas 11.** This malware infects ELF files, and provides already known polymorphic variants for evading the traditional malware detectors.

**Woolien.** It has three modes and provides a remote shell to the intruder. The idea is to open the host to the network attacks.

**Rape.** It infects the Linux Installer files namely RPMs. The infected RPM files run the payload at the host where they are installed.

**Siilov.** It modifies the procedure link table to redirect the `exeve` calls to its own desired code. If the malware is run by the root user, the `init` becomes infected as well which in turn infects all the processes. Data segment is used for carrying the malicious code.

**Silvio.** This virus uses the first page of the text segment for infection, and in order to achieve parasitism, the malware extends the text segment backwards to make room for the extra code.

**Tlb.** It is an ELF file infector and uses encryption and mutation to evade malware detectors.

**Vlpi.** It includes a scanner for finding vulnerable benign files. Rest of the malware source is identical to other ELF infectors.

**5.3.3 Malware Test & Training Dataset**

We have evolved hundreds of malware variants using our malware evolution engine. However, a limited number of these malware have been used to evaluate the static and dynamic malware detection schemes of our framework. Due to the scant availability of ARM Linux benign files on OpenMoko, EST uses 430 malware samples i.e. 43 variants of each malware and 422 benign files (taken from the OpenMoko’s system directories `/bin`, `/sbin` and `/usr/bin/`). PRT, on the other hand, is evaluated using 100 benign and 100 malware samples i.e. 10 variants of each malware. It is also necessary to dump the process control blocks (PCBs) for the execution of benign and malware files for the training of classifiers and testing procedures. It is important to note that once a malware executes, we need to reinstall the OS on OpenMoko handset. We have used a 10-fold cross validation scheme for all experiments conducted and reported in Section 5.5 and 5.7. We prepare two types of training and test datasets as explained in the following subsections.

**5.3.3.1 Singleton Test & Training-set**

All samples of an individual malware and equal number of benign samples are separated to construct the Singleton test-set. For instance, all samples of Dataseg malware and an equivalent number of benign samples are used to form the Singleton test-set. The training-set is composed of malware samples excluding the entire samples of Dataseg malware and equivalent number of benign samples i.e. all sample of [Rape, Sick, Silvio, Siilov, Woolien, Tlb, Vlpi, Kaiowas10 and Kaiowas11] malware are combined with equal number of benign samples to form the training set. In this
way, 10 different test and training sets are created, to demonstrate the scenario of zero-day malware detection.

5.3.3.2 Existere Test & Training-set

Existere test-set is composed of 10% samples of all 10 malware and equivalent number of random benign samples. Rest of the malware and benign samples are combined to create an existere training set. It illustrates the scenario in which training set contains some polymorphic samples of the malware that need to be detected.

Now we briefly describe the architecture of our hybrid framework for malware detection on smartphones.

5.4 A Hybrid Architecture of Malware Detection Framework

In this section, we present the architecture of our two-tier framework shown in Figure 5.2. Both the static and dynamic malware detection techniques in the framework are composed of different sub-modules which are discussed in their relevant sections.

Figure 5.2: A hybrid architecture for malware detection on smartphones

5.5 ELF Structural Tracer: Case Study

PRT \(^1\) mines the structural information from executables on ARM Linux platform to detect malicious files. Major components of EST (see Figure 5.2) are described in the following subsections.

\(^1\)Initial work on this approach is published as “ELF-Miner” [Shahzad and Farooq, 2012] that detects zero-day malicious executables on Linux x86 platform.
5.5 ELF Structural Tracer: Case Study

5.5.1 EST prologue

As the EST kernel module is started, Prologue section proceeds with the following steps executed in sequential order.

1. EST framework preloads the library which consists of a bunch of customized \textit{exec-xx} functions. In Linux, every process is invoked through \texttt{execve} system-call.

2. Whenever, an executable file is started by the user on OpenMoko handset, \texttt{execve} is called by the Linux kernel. EST framework intercepts it and instigates the customized \texttt{cust_execve} function from the preloaded library. It also captures the path of the executable file available in \texttt{execve} parameters list.

3. Customized library function \texttt{cust_execve} opens the Executable and Linkable File (ELF), extracts and validates the ‘\textit{ELF}’ signature which resides in the first 4 bytes of this file. Afterwards, structural features are extracted, preprocessed and classified by EST framework. If EST framework marks any file as malicious, its execution is aborted and control is transferred back to the kernel instead of \texttt{execve} system-call which normally launches the process.

5.5.2 ELF Structural Features Extractor

Initially, 383 features are extracted by \texttt{cust_execve} function from the headers of ARM Linux ELF. A brief forensic analysis is performed to prove that structural features are enough to discriminate a benign and a malware file. The description of ELF mined feature set is eliminated from this chapter but an interested reader can find them in [Shahzad and Farooq, 2012][TI-Standards, 1993].

5.5.3 Forensic Analysis

A succinct forensic analysis is provided to strengthen the argument that the structural information in ARM Linux executables’ headers can be used to discriminate the malware files from benign ones’. We use renowned information-theoretic measures – Resistor-Average divergence (RAD) [Johnson and Sinanovic, 2001b] – to measure the potential difference between headers of benign and malware files (only a few headers are analyzed). RA divergence is defined as:

\[
RA(p, q) = 1/(\frac{1}{KL(p||q)} + \frac{1}{KL(q||p)})
\]  

(5.1)

In equation (5.1), Kullback-Leibler (KL) distance [Cover and Thomas, 2006] is another information-theoretic measure, – used to measure the dissimilarity between two probability distributions \( p(x) \) and \( q(x) \) (equation 5.2). To avoid zero and infinite values, we add \( \epsilon = 2^{-52} \) in both distributions (it is distance from 1.0 to next largest double precision number).

\[
(KL(p||q)) = \sum_{x} p(x) \log \frac{p(x)}{q(x)}
\]  

(5.2)

\textbf{ELF Header Examination.} The RAD for 16 parameters of ELF header structure is plotted in Figure 5.3(a). Interestingly, five fields are notably different
5. A Hybrid Framework for Malware Detection on Smartphones using ELF Structural & PCB Runtime Traces

Figure 5.3: Difference between benign and malware headers

in both benign and malware files i.e. \texttt{e\_type}, \texttt{e\_entry}, \texttt{e\_ehsize}, \texttt{e\_shnum} and \texttt{e\_shstrndx}. These fields consists of different headers’ indices and sizes (numeric values). Remember larger RAD values depict considerable difference in benign and malware executables. Other fields hold constant or zero values and consequently, their RAD is either zero or negligible.

**Symbol Table.** The plot of RAD for 17 different categorical fields of symbol table is shown in Figure 5.3(b). It is observed that, apart from processor specific categories i.e. \texttt{stb\_hiproc}, \texttt{stb\_loproc}, \texttt{sst\_hiproc} and \texttt{stt\_loproc} which are not found in benign and malicious executables on ARM Linux, a remarkable disparity can be seen between symbol table’s entries. Most of the executable on ARM Linux are stripped binaries i.e. they don’t have a symbol table while malware files contain symbol table entries.

**Dynamic Symbol Section.** The RAD is plotted for 17 fields of dynamic symbol section in Figure 5.3(c). It is observed that some categorial fields have high RAD values e.g. \texttt{total}, \texttt{stb\[global, weak\]}, \texttt{stt\_ntype}, \texttt{stt\_obj\[global, weak\]}, \texttt{stt\_func\[global, weak\]}. Most of these discriminating fields represent different categories of dynamic symbols that are used by benign and malware programs to present global & local objects and functions.

**Relocation Symbol Section.** The graph of RAD for relocation sections’ fields is shown in Figure 5.3(d). Only one relocation type \texttt{relr386pc32} is present in relocation section plot. This relocation type supports PC-relative addressing which is commonly used by the malware programs. However, benign executables commonly use absolute addressing. This feature is also beneficial for distinguishing between
5.5 ELF Structural Tracer: Case Study

benign and malicious files.

Figure 5.4: DWT implementation using Daubechies-1 wavelet for feature selection

5.5.4 Pre-Processing Filters

5.5.4.1 Redundant Feature Removal (RFR)

In order to eliminate the features which are either constant or show significantly larger variance, we apply RFR filter. RFR truncates the EST dataset and removes the features that have constant values or show larger variance. For instance, when we applied RFR filter to EST Singleton and Existere datasets. It removed 239, 114 features and 144, 269 features remained in Singleton and Existere datasets respectively. The reason for the removal of high percentage of the features is that a lot of fields in ELF headers have constant or null values.

5.5.4.2 Dimensionality Reduction

In order to reduce the classification overhead of classifiers, we apply dimensionality reduction technique called Discrete Wavelet Transform (DWT). Although it has been extensively used in image compression, it is rarely utilized for the detection of malware. In one dimensional DWT, the original signal is divided into two equivalent segments: (1) approximation component which contains the most prominent and relevant information, and (2) detailed component which contains fine and minute details of the signal. The decomposition process of wavelet is initiated and the original signal (ELF structural features’ array) $x[n]$ is passed through a half-band lowpass digital filter with impulse response $h[n]$ which is equivalent to convolution of the signal with the impulse response of the filter. Mathematically, it can be written as:

$$x[n] * h[n] = \sum_{k=-\infty}^{\infty} x[k] . h[n-k]$$

Passing the signal through a lowpass filter halves its resolution. However, scale remains unchanged. The signal is sub-sampled by 2 since half of the samples are redundant. As a result, the scale is doubled. We can express it as:
5. A Hybrid Framework for Malware Detection on Smartphones using ELF Structural & PCB Runtime Traces

\[ y[n] = \sum_{k=-\infty}^{\infty} h[k].x[2n - k] \quad (5.4) \]

Finally, signal \( x[n] \) is passed through a highpass filter \( g[n] \) and a lowpass filter \( h[n] \) simultaneously. After the filtering process, half of the samples can be eliminated by Nyquist’s rule. Moreover, the signal can be sub-sampled by 2 effectively discarding every other sample. DWT then analyzes the signal at different frequency bands with different resolutions and decomposes it into the approximation (output from lowpass filter (Equation 5.6)) and detailed (output from highpass filter (Equation 5.5)) sub-band.

\[ y_{\text{high}}[k] = \sum_{n} x[n].g[2k - n] \quad (5.5) \]
\[ y_{\text{low}}[k] = \sum_{n} x[n].h[2k - n] \quad (5.6) \]

However, filters are usually not ideal half-band. Therefore, perfect decomposition cannot be achieved. However, under some specific conditions, it is possible to find perfect filters e.g. Daubechies’ wavelets which provide reasonable accuracy. Now, the following equation can be used to synthesize the approximation and detailed coefficients of DWT.

\[ x[n] = \sum_{k=-\infty}^{\infty} \left( y_{\text{high}}[k].g[-n + 2k] + y_{\text{low}}[k].h[-n + 2k] \right) \]

DWT of a sample benign and malware scaling function (structural trace) along with their approximation and detailed subbands using Daubechies-1 wavelet are plotted in Figure 5.4(a) and 5.4(b). Using DWT, we select 72 and 134 prominent features (approximation coefficients) from 144 and 269 features of EST Singleton and Existere training and test-sets.

5.6 EST Classification & Results

In this section, we discuss the malware detection-accuracy using the prominent features of ELF structural traces obtained through DWT. We have used two different paradigms of classifiers’ training and testing i.e. (a) Singleton and (b) Existere test and training set as discussed in Section 5.3.3. A stratified 10-fold cross validation strategy is used. Both the datasets are divided into 10 folds. A fold of Singleton dataset consists of 43 different sample of an individual malware and 42 randomly selected benign samples. Other 9 folds are used for classifiers training and the remaining 1 fold is used for testing. Similarly, a fold of Existere dataset consists of 43 i.e. 10% different samples of all 10 malware and 42 randomly selected benign samples, 9 folds are used for classifiers training and the remaining 1 fold is used for testing.

We have used 4 renowned machine-learning-classifiers for comparative analysis of results. During the classification, we have investigated the following issues: (1) overall accuracy of classifiers for zero-day polymorphic malware detection, (2) detecting the new polymorphic variants of malware that already exist in a training
set, (3) computation of processing overheads of feature extraction, pre-processing, training and testing modules of ELF structural tracer and (4) the robustness of the ELF structural traces to evasion attempts.

\[
DR = \frac{TP}{TP + FN} \quad (5.7)
\]

\[
FAR = \frac{FP}{FP + TN} \quad (5.8)
\]

\[
DA = \frac{(TP + TN)}{(TP + TN + FP + FN)} \quad (5.9)
\]

In a typical two-class problem, such as malicious executable detection, the classification decision of an algorithm may fall into one of the following four categories: (1) true positive (TP), classification of a malicious executable as malicious, (2) true negative (TN), classification of a benign executable as benign, (3) false-positive (FP), classification of a benign executable as malicious and (4) false-negative (FN), classification of a malicious executable as benign. The detection-accuracy of proposed system is reported using three separate metrics: (1) detection rate (DR), (2) false-alarm-rate (FAR), and (3) detection-accuracy (DA). These metrics are defined in equations 5.7, 5.8, 5.9.

### 5.6.1 Zero-day polymorphic malware detection

The polymorphic malware variants’ detection-accuracy of the four classifiers at zero-day is reported in Table 5.1 using ELF structural traces and Singleton test and training sets. Each fold of Singleton test-set contains different polymorphic variants of a previously unknown malware. In addition, the reported metric values are an average of 10 folds for each classifier to avoid the repetition of similar results of folds. It can be deduced from the Table 5.1 that DR, FAR and DA of both JRip and J48 equal 99.83%, 0% and 99.41% respectively. Only a few variants of malware kaiowas11 are misclassified. Other classifiers have almost identical accuracy. Therefore, we can infer that EST is proficient to detect zero-day polymorphic malware executables.

Table 5.1: Overall classification results with Singleton training and test-set

<table>
<thead>
<tr>
<th>Features &amp; Classifiers</th>
<th>Approximation Wavelet Subband Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Headers</td>
<td>JRip</td>
</tr>
<tr>
<td>Wavelet-Features</td>
<td>Filtered-ELF-Traces</td>
</tr>
<tr>
<td>Classifiers</td>
<td>IBK</td>
</tr>
<tr>
<td>DR</td>
<td>98.83</td>
</tr>
<tr>
<td>FAR</td>
<td>0</td>
</tr>
<tr>
<td>DA</td>
<td>99.41</td>
</tr>
</tbody>
</table>

Table 5.2: Overall classification results with Existere training and test-set

<table>
<thead>
<tr>
<th>Features &amp; Classifiers</th>
<th>Approximation Wavelet Subband Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Headers</td>
<td>JRip</td>
</tr>
<tr>
<td>Wavelet-Features</td>
<td>Filtered-ELF-Traces</td>
</tr>
<tr>
<td>Classifiers</td>
<td>IBK</td>
</tr>
<tr>
<td>DR</td>
<td>100</td>
</tr>
<tr>
<td>FAR</td>
<td>0</td>
</tr>
<tr>
<td>DA</td>
<td>100</td>
</tr>
</tbody>
</table>

### 5.6.2 Existere polymorphic malware detection

We perform another experiment using the Existere training and test-set to detect the new polymorphic variants of existing malware using the JRip and J48 classifiers trained on the ELF structural traces of already available variants of the similar
5. A Hybrid Framework for Malware Detection on Smartphones using ELF Structural & PCB Runtime Traces

malware. The malware detection-accuracy metrics of all the four classifiers are reported in Table 5.2. The classifiers achieve DR, FAR and DA of 100%, 0% and 100% respectively. The most important point to note is that not even a single variant of any malware remained undetected. Therefore, we conclude that EST can successfully detect the new variants of any already known malware.

5.6.3 Processing Overheads

We now analyze the processing overheads of EST. The overheads include the timing intervals for features extraction, pre-processing filters (RFR & DWT), classifier training and testing. On average, features extraction and pre-processing times per file are 20.41 and 4 ms respectively. The training times of JRip and J48 are 0.817 and 0.286 ms respectively. It should also be noted that training time is not a critical issue since training is carried out on the loading of kernel module and then on hourly basis. Similarly, the testing time of both JRip and J48 is 14 µs (including context switching time). So overall processing overhead of EST using JRip or J48 classifiers is 38.41ms per executable file i.e. the sum of the features extraction, pre-preprocessing and testing time.

5.6.4 EST features’ robustness analysis

We now analyze the robustness of our EST features set in a specific scenario where malware writers synchronize different structural header of their malware with that of benign executable. To demonstrate the header spoofing, we gradually replaced malware files (one of the malware samples in Singleton test-set) headers with benign headers and recalculated the corresponding DR. It is interesting to analyze that all classifiers maintain 98 − 99% DR even when more than 70% features are forged. It is because of the reason that randomly selected headers are replaced gradually. However, some the remaining features have higher classification potential. This proves that EST is also robust to evasion attempts of malware writers – because multiple headers contain the features with high classification potential.

5.7 PCB Runtime Tracer: A Case Study

Whenever a malicious executable passes through EST malware detection filters, cust_execve transfers control to execve system-call and processes are launched by the kernel. Now the runtime component of our framework activates and detects the malware during its execution. The architecture of the in-execution dynamic malware detection technique, PRT ¹, is presented here which extracts the runtime information (blueprint) of processes from ARM Linux kernel’s PCBs to detect the malicious processes. The zero-day malware in-execution and polymorphic variants of malware which have minor differences in their execution patterns can also be detected by PRT. Moreover, it is notable that, during execution, the PCB runtime trace of an unpacked malware and its packed variants are almost identical. Since every packed malware is unpacked before the start of its execution, PRT can also

¹Initial work on this approach is published in [Shahzad et al., 2011b] and [Shahzad et al., 2011a] that dynamically detects malicious processes during execution on x86 Linux platform.
effectively detect packed variants of malware. PRT consists of the following two components: (1) PRT features extractor, and (2) classifier (see Figure 5.2).

## 5.7 PCB Runtime Tracer: A Case Study

### 5.7.1 PRT Features extractor

We have implemented the PRT component as a kernel module on OpenMoko smartphone. Its features logger module extracts 14 parameters (finally chosen of 118 parameters, see off-line analysis for details) from PCBs (task_structures) of ARM Linux kernel for the newly launched process. The features tracer periodically extracts the parameters of task_struct with a resolution of 10 ms. The prominent parameters include number of page frames and page tables, volunteer and in-volunteer context switches, number of page table locks and page faults, CPU time of process in kernel and user mode etc. In the following subsections, we establish the reliability of these 14 parameters to discriminate the benign and malicious processes at runtime.

#### 5.7.1.1 PCB runtime features - off-line analysis

For the off-line analysis, initially we log 118 parameters from PCBs of OpenMoko’s ARM Linux kernel by executing the individual processes with a time resolution of 10 ms. Linux kernel maintains a doubly circular linked list of processes (see Figure 5.2) and adds a new task_struct structure whenever a new process is launched. A global pointer, current, is used to gain access to the task structure of an executing process. Then, by tracking the circular link list, any process can be accessed. The kernel module tracks the desired process by its name in the list of processes. As it finds the process, it dumps its 118 task_struct parameters in a text file. For off-line training and analysis, we dump at most 10000 samples for each process (1 minute and 40 seconds execution dump). The processes that complete their execution – finish earlier – obviously dumped samples would correspond to the duration of their execution. The feature extractor module dumps PCB parameters for 100 benign and malicious processes.

#### 5.7.1.2 Forensic investigation

We conduct a forensic investigation to prove that the execution behavior of benign and malicious processes is different. Furthermore, we also show that parameters of PCBs can be used to distinguish among benign and malicious processes. To this end, we have compared two parameters of task_struct: (1) number of pages table entries and (2) process’s tick count in user mode using four different benign and malicious processes.

**Execution Patterns of Benign Processes.** The execution patterns of two benign processes are plotted in Figure 5.5(a) and 5.5(b). It can be inferred from Figure 5.5(a) that for parameter nr_pures, – unzip utility shows an linearly increasing pattern till 200 units and 1100 instances. On the other hand, cp exhibits linear pattern for page table entries up to 40 units and 4200 instances. Execution patterns of the time and battery utilities is a straight line for utime parameter up to 3/65 units and 3000/8000 instances respectively (Figure 5.5(b)).

---

1Each parameter instance is logged with a resolution of 10 ms
5. A Hybrid Framework for Malware Detection on Smartphones using ELF Structural & PCB Runtime Traces

Figure 5.5: Difference between benign and malicious processes execution patterns
### Table 5.3: Description of the fields constituting PCB runtime trace

<table>
<thead>
<tr>
<th>No</th>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>task→active_mm→map_count</td>
<td>No. of memory regions of a process</td>
</tr>
<tr>
<td>2</td>
<td>task→active_mm→page_table_lock</td>
<td>Needed to traverse &amp; manipulate the page table entries</td>
</tr>
<tr>
<td>3</td>
<td>task→active_mm→hiwater_rss</td>
<td>Max no. of page frames ever owned by a process</td>
</tr>
<tr>
<td>4</td>
<td>task→active_mm→hiwater_vm</td>
<td>Max no. of pages appeared in memory region of process</td>
</tr>
<tr>
<td>5</td>
<td>task→active_mm→total_vm</td>
<td>Size of process’s address space in terms of no. of pages</td>
</tr>
<tr>
<td>6</td>
<td>task→active_mm→shared_vm</td>
<td>No. of pages in shared file memory mappings of process</td>
</tr>
<tr>
<td>7</td>
<td>task→active_mm→exec_vm</td>
<td>No. of pages in executable memory mappings of process</td>
</tr>
<tr>
<td>8</td>
<td>task→active_mm→nr_pdirs</td>
<td>No. of page tables of a process</td>
</tr>
<tr>
<td>9</td>
<td>task→utime</td>
<td>Tick counts of a process that is executing in user mode</td>
</tr>
<tr>
<td>10</td>
<td>task→stime</td>
<td>Tick counts of a process in the kernel mode</td>
</tr>
<tr>
<td>11</td>
<td>task→nvosw</td>
<td>Number of volunteer context switches</td>
</tr>
<tr>
<td>12</td>
<td>task→nvivosw</td>
<td>Stores the no. of in-volunteer context switches</td>
</tr>
<tr>
<td>13</td>
<td>task→alloc_lock.raw_lock.slock</td>
<td>Used to lock memory manager, files and file system etc</td>
</tr>
<tr>
<td>14</td>
<td>task→fs→count</td>
<td>fs_struct’s usage count to indicate the restrictions</td>
</tr>
</tbody>
</table>

### Execution Patterns of Malware Processes

The execution patterns of two malicious processes are plotted in Figure 5.5(c) and 5.5(d). It can be observed from Figure 5.5(c) that vlpi malware, after 2500 instances, has an linearly increasing pattern till 2900 units and 4000 instances. On the other hand, kaiowas10-4 malware shows a linear pattern up to 75 units and 2600 instances. The execution behaviors of the two malicious processes and benign processes are significantly different from each other in terms of page table entries. Figure 5.5(d) shows that the execution pattern of two malicious processes, dataset-5 and kaiowas10-7, in terms of utime, shows a gradual increase up to 12/7000 units and 3000/3400 instances and are much different from the plots of benign processes (see Figure 5.5(b)). The results clearly demonstrate that the parameters of process control blocks of ARM Linux kernel can be used to identify the malicious processes.

### 5.7.1.3 Features Selection

PCB runtime features are preprocessed in two steps. In the first step, we identify and eliminate fields which are irrelevant to the runtime behavior of a process i.e. constant parameters, process identifiers and flag fields (bit combinations) etc. This is an important phase since these features can misguide a machine-learning-classifier during the training process. In the case of OpenMoko PCBs, 23 fields are different kinds of offsets, 9 are bit combination (flags) and other 51 parameters contain constant or zeros values. Therefore, a total of 83 features are eliminated in the initial filtration leaving us with 35 parameters only. In the second step, time series mean is calculated for all the remaining 35 parameters of benign and malicious processes using the following equation:

\[
M_{tsm} = \mu_{t,pr-idx} = \frac{1}{N_{proc}} \sum_{p_{num}=1}^{N_{proc}} x_{t,p_{num},pr-idx} 
\]

(5.10)

\[
t = 1 \rightarrow t_{term}, pr-idx = 1 \rightarrow 35
\]

where \( t \) represents the time instance ranging from \( 1 \) to \( t_{term} \) (termination time of a process in milliseconds), \( p_{num} \) is the processes’ identification number, \( N_{proc} = 100 \) is the total number of benign and malicious processes, \( \mu_{t,pr-idx} \in M_{tsm} (M_{tsm} \) are the time series means of parameters \( pr-idx = 1 \) to \( 35 \)). \( M_{tsm} \) is used to identify distinct fields in the dataset of benign and malicious processes. Furthermore, time series
variance \((V_{tsv})\) is also calculated and analyzed. If the parameters have high variance, they will mislead the classifier causing high false-positives and false-negatives respectively. The variance is computed using the following equation:

\[
V_{tsv} = \sigma^2_{t,pr-idx} = \frac{1}{N_{proc}} \sum_{p_{num}=1}^{N_{proc}} (x_{t,p_{num},pr-idx} - \mu_{t,pr-idx})^2 \quad (5.11)
\]

As a next step, the coefficient of variance \((cv_{t,pr-idx})\), a normalized measure of dispersion of the parameters probability distribution, is computed using:

\[
cv_{t,pr-idx} = \frac{\sigma_{t,pr-idx}}{\mu_{t,pr-idx}} \quad i = 1 \rightarrow t_{term}, pr-idx = 1 \rightarrow 35 \quad (5.12)
\]

where \(t\), \(p_{num}\), \(\mu\), \(\sigma\), \(t_{term}\) and \(pr-idx\) denote the same quantities as in Equations (5.10) and (5.11). \(cv_{t,pr-idx}\) denote the coefficient of variance of parameter \(pr-idx\) at time instant \(t\). Remember features with large value of \(cv_{t,pr-idx}\) can mislead the classifiers’ training process and adversely affect the accuracy.

Now we discuss sequential steps for PCB runtime trace’s formal features’ selection. The time series mean of two parameters, page_table_lock and total_vm, are shown in Figure 5.6(a) and Figure 5.6(c) respectively. It can be observed that the value of \(M_{tsm}\) for both parameters is significantly different for benign and malicious processes. Both parameters pass through first filter. Their time series variance is plotted in Figure 5.6(b) and Figure 5.6(d). It is clearly visible that plots for benign and malicious processes differ from each other. In the third step, time series coefficients of variance are plotted (see Figures 5.6(e) and 5.6(f)). The plot of \(cv_{t,pr-idx}\) for both parameters lies between 0.5 – 3.0 for benign as well as malicious processes. An important conclusion inferred from these results is that both parameters should be included in PCB runtime trace.

The time series mean and variance for an additional parameters fpu-counter, is shown in Figure 5.6(g) and 5.6(h) respectively. Since both mean and variance is zero for benign and malicious processes, its coefficient of variance becomes undefined. As a result, it is not suitable parameter for inclusion in PCB runtime trace. Remember, if the value of coefficient of variance is less than 3, the distribution is either normal/gaussian and 95.45% of its values lie between \(\mu \pm 3\sigma\). Based on this analysis, the following rule set is evolved for PCB runtime trace:

1. The time series mean and variance of a parameter should be significantly different for both benign and malicious processes.
2. The coefficient of variation should be less than 3.
3. Finally, due to normal/gaussian distribution, 95.45% of its values should be within the interval \(\mu \pm 3\sigma\).

The above derived rules are iteratively applied to the rest of the 35 time series parameters. Consequently, 21 features are disqualified leaving us with 14 parameters tabulated in Table 5.3. After the completion of feature selection process, the next step is to classify the PCB runtime traces’ dataset of processes on OpenMoko smartphone. We have already established in [Shahzad et al., 2011b] that machine-learning-classifiers J48 and JRip are appropriate for PCB mined noisy datasets. They also provide high detection accuracy with low false-alarm-rate. Furthermore, their training and testing durations are also suitable for real time deployment.
5.7 PCB Runtime Tracer: A Case Study

Figure 5.6: Plots of various statistical aspects of short listed parameters

(a) Accepted Parameter page_table_lock showing difference in time series mean for benign and malicious processes

(b) Accepted Parameter page_table_lock showing low variance

(c) Accepted Parameter total-vm showing difference in time series mean for benign and malicious processes

(d) Accepted Parameter total-vm showing low variance

(e) Accepted Parameter page_table_lock showing low coefficient of variation

(f) Accepted Parameter page_table_lock showing low coefficient of variation

(g) Rejected Accepted Parameter fpu-counter showing low difference in time series mean for benign and malicious processes

(h) Rejected Accepted Parameter fpu-counter showing low difference in time series variance for benign and malicious processes
5.8 PRT Classification and Results

Now we evaluate the malware detection-accuracy of PRT using the short-listed features and the metrics defined in Section 5.6. The following points are emphasized in this discussion: (1) overall accuracy of the classifiers for zero-day polymorphic malware detection, (2) the impact of multi-window processing on the classification accuracy, (3) detection of new polymorphic variants of malware which already exist in the training set, (4) packed malware detection using PRT, (5) measurement of the processing overheads of logging, training and testing of different modules and (6) the robustness of PRT to evasion attempts.

The accuracy and significance of PRT is evaluated in detecting malicious processes using machine-learning-classifiers J48 and JRip. For this purpose, we have used two different datasets: (1) Singleton, (2) Existere, test and training sets. The composition of these datasets is discussed in section 5.3.3. A stratified 10−fold cross validation strategy is used. Both datasets are divided into 10 folds where a fold of Singleton test-set consists of the traces of 10 different variants of a malware and runtime traces of 10 benign processes selected randomly. The runtime traces of benign and malicious processes, 90 each, are used for training purpose. Similarly, a fold of Existere test dataset consists of 10% different samples of all 10 malware and runtime traces of 10 randomly selected benign processes – remaining runtime traces of benign and malicious processes are used for classifiers’ training.

It should also be noticed that all benign and malware processes should have equal ratio of instances in the training set to eliminate the bias towards the longer processes (shorter duration processes complete before 10 sec). To remove any implementation related bias of the classifiers, WEKA’s Knowledge Flow Tool is used in this analysis [Witten and Frank, 2002]. We now discuss the issues raised earlier in this section.

5.8.1 Zero-day - polymorphic malware Detection

The malware detection-accuracy of the two classifiers is reported in Table 5.4 in terms of average values. The ‘window size’ delineate the number of considered PCB runtime trace instances for the sampling of each process before making a decision. In 100 instances (1000ms) window (or 10 windows of 10 instances each), the classifiers take first 100 instances of each parameter in PCB runtime trace to decide if the process is malicious or benign. For instance, if 51% instances are declared malicious in a window, the process is declared as malicious. The decision is deferred to the next window if the votes are equal. It is clear from the results listed in Table 5.4 that DR, FAR and DA of JRip, in case of a previously unknown and polymorphic malware are 92.0%, 0% and 96.0% respectively in a window size of 100 instances. All previously unknown malware variants, with respect to the training set, are detected correctly except the variants of dataseg malware. For J48, these parameters are 3-7% inferior. To the best of our knowledge, no existing dynamic and in-execution malware detection system on smartphones has such a high DA and low FAR.

5.8.2 Multi-window processing & detection-accuracy

We repeat the above experiment with multiple windows (10 windows) of 100 instances each for dataseg malware and its variants using both JRip and J48 classifiers. The results are shown in Figure 5.7. Its can observed from Figure 5.7(a)
5.8 PRT Classification and Results

![Accuracy plot for JRIP and J48 classifiers for multiple windows processing](image)

Figure 5.7: Accuracy plot for JRIP and J48 classifiers for multiple windows processing

that accuracy gradually increases from 4th to 10th window. On completion of 10th windows, all variants of the dataset malware are detected i.e. in 10 sec (10000 ms) leading to 100% DA and 0% FAR (see Table 5.4 for complete results). The only drawback of processing multiple windows is the high processing overhead. It is also important to note that J48 could not improve overall accuracy even by processing multiple windows.

5.8.3 Existere malware - new variants detection

To detect the new polymorphic variants of malware e2using the JRip and J48, a new experiment is conducted using the Existere dataset classifiers trained on the PCB runtime traces of a similar malware. The results are tabulated in Table 5.5. It can be seen that both classifiers achieve DR, FAR and DA of 100%, 0% and 100% respectively when a window of 100 instances (or 10 windows, each of size 10) are processed. Therefore, we conclude that PCB runtime traces are handy in detecting new variants of a known malware.

5.8.4 Packed Malware detection using PRT

A packer software encrypts and compresses the code and resource sections of an executable file using its encryption and compression libraries [Yan et al., 2008]. The packer produces multiple variants of a single executable file in each packing process. Moreover, the polymorphic engines of the state-of-the-art packing tools add some protection mechanisms against reverse engineering and debugging processes. On the other hand, in order to restore original binary code, the packer stub unpacks the executable file before the start of its execution. Rest of the execution of both files is identical in the kernel. Therefore, we conclude that PCB runtime trace of both packed and unpacked binary files are identical except the unpacking duration of packed files. Hence, PRT is equally competitive to detect packed and unpacked malicious files on OpenMoko’s ARM architecture.
Table 5.4: Accuracy results of J48 and JRip on each of the 10 folds using Singleton training and test-set scheme

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Time</th>
<th>Metrics</th>
<th>Fold1</th>
<th>Fold2</th>
<th>Fold3</th>
<th>Fold4</th>
<th>Fold5</th>
<th>Fold6</th>
<th>Fold7</th>
<th>Fold8</th>
<th>Fold9</th>
<th>Fold10</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>J48</td>
<td>100 Ins (1000 ms)</td>
<td>DR(%)</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>90</td>
<td>00</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>79.0</td>
</tr>
<tr>
<td></td>
<td>1000 Ins (10000 ms)</td>
<td>FAR(%)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DA(%)</td>
<td>50</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>95</td>
<td>50</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>89.50</td>
</tr>
<tr>
<td>JRip</td>
<td>100 Ins (1000 ms)</td>
<td>DR(%)</td>
<td>20.0</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>92.0</td>
</tr>
<tr>
<td></td>
<td>1000 Ins (10000 ms)</td>
<td>FAR(%)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DA(%)</td>
<td>60</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>96.00</td>
</tr>
<tr>
<td>JRip</td>
<td>1000 Ins (10000 ms)</td>
<td>DR(%)</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FAR(%)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DA(%)</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100.00</td>
</tr>
</tbody>
</table>

Table 5.4: Accuracy results of J48 and JRip on each of the 10 folds using Singleton training and test-set scheme.
Table 5.5: Accuracy results of J48 and JRip on each of the 10 folds on Existere training and test-set scheme

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Time</th>
<th>DR(%)</th>
<th>FAR(%)</th>
<th>DA(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>J48</td>
<td>100 Ins (1000 ms)</td>
<td>100</td>
<td>0.00</td>
<td>100</td>
</tr>
<tr>
<td>JRip</td>
<td>100 Ins (1000 ms)</td>
<td>100</td>
<td>0.00</td>
<td>100</td>
</tr>
</tbody>
</table>
5.8.5 Processing overheads

In a real-time malware detection technique, it is essential to measure the processing overhead of each component of the system. The processing overhead of PRT is the sum of feature logging and testing times. The training time of classifier is not critical because it is only carried out at the start of kernel module and then on hourly basis. The PRT feature extractor consumes $80 \mu s$ on OpenMoko for each instance including the context switching time of $8 \mu s$. Both J48 and JRip classifiers take 42 and 55 ms for the training process respectively. If we also include their testing time i.e. 50 and 110 $\mu s$ respectively, the total time of J48 and JRip per instance equal 130 and 190 $\mu s$ respectively. The results are very intriguing because, using 100 instances window size, the processing time of detecting a malicious process is approximately 1013 ms (1.3% overhead for each window).

5.8.6 Robustness to evasion attempts

In order to initiate the robustness analysis of 14 features of PCB runtime trace, different features in the trace set of a malicious process are replaced with the corresponding parameters of an identical benign process. This is a cogent and logical scheme for the emulation of a benign process as a malicious one and vice versa. The impact of this counterfeiting is measured and analyzed on the overall accuracy of PRT technique. All the reported results use a window size of 100 and one fold of Singleton dataset. It can be seen that, when 4 features are forged, only 8% more variants of malicious processes are misclassified. With 7 forged features, 15 malicious processes are misclassified as benign. The accuracy degrades further with 9 forged parameters. However, it is still comparable with the existing state-of-the-art behavioral systems on smartphones.

The above experiments are performed by assuming that the PCB runtime traces of both benign and malicious processes for OpenMoko’s ARM architecture are already known. However, it must be remembered that parameters of PCB are highly dependent on a specific hardware configuration of the smartphones i.e. RAM, processor, paging algorithms, stack and heap managers etc. Therefore, they must be collected for every independent configuration of the host. If PRT component is deployed at any other Linux based smartphone, these parameters’ values would be changed accordingly. As a result, a malicious software writer has to estimate these parameters through hooks which in turn need administrative privileges. We have elaborated this point in [Shahzad et al., 2011b] that kernel rootkits are critical threats to OS because they can hide themselves and access kernel PCBs. Researchers have proposed different solutions (see for instance [Hund et al., 2009] and [Riley et al., 2008]) for the protection of kernel and PCB. To conclude, PCB runtime tracer can be deployed in conjunction with any of the existing security solution for protecting kernel from rootkits.

5.9 Conclusions and Future Work

In this chapter, we have presented a hybrid framework for ARM-Linux based smartphones which detects zero-day, polymorphic and repacked malware while residing on disk and during their execution. We have utilized cross-compiled x86-Linux-malware
for training and testing of the proposed hybrid framework. The empirical results demonstrate that EST component that detects zero-day malware, achieved more than 99% DA with 0% FAR and 100% DA with 0% FAR, respectively, on Singleton and Existere classification schemes. On the other hand, PRT component that detects polymorphic variants of malware, achieved 96% DA with 0% FAR and 100% DA with 0% FAR, respectively, on Singleton and Existere classification schemes. We also demonstrated the resilience of our framework against structural and run-time evasion attempts. Moreover, the malware detection time & processing overhead of our framework is negligible. Keeping in view of these results, we can conclude that our framework can detect malicious executables and processes in realtime on Linux based smartphones. In future research, we intend to evaluate the effectiveness of our PRT security component for other Linux based smartphone operating systems especially Android.
Chapter 6

TStructDroid: Realtime Malware Detection using Time-series Analysis of PCB on Android

6.1 Introduction

As the smartphone devices have become a basic necessity and their use has become ubiquitous in recent years, the malware attacks on smartphone platforms have escalated sharply. As an increasing number of smartphone users tend to use their devices for storing privacy-sensitive information and performing financial transactions, there is a dire need to protect the smartphone users from the rising threat of malware attacks. In this chapter, we present a realtime malware detection framework for Android platform that applies information-theoretic tools on the logged time-series data, obtained from the footprint available in the process control blocks of the operating system kernel of an executing process to build a model that can discriminate the execution traces of benign and malware processes.

Smartphones have become a critical enabler for providing connected m-services – including but not limited to m-health, m-government, m-payment, and m-banking – and therefore their penetration has significantly increased in the mobile market in the previous 3 to 4 years. It is reported that 419.1 million mobile devices were in the market at the beginning of year 2012 that were estimated to become 645 million at the end of year 2012\(^1\) [Gartner, 2012].

The popularity of smartphones has generated an interest to understand its usage dynamics. Recently, a survey conducted by Kaspersky Labs [Product-News, 2012] shows that users primarily use smartphones for: (1) storing personal information and documents (16%), (2) Internet surfing (62%), (3) email communication (53%), and (4) social networking (47%). The above-mentioned usage is possible because smartphones are like computing devices with their own operating system. Another study by Gartner [Gupta et al., 2012] revealed that smartphones, running Google’s Android OS, have captured an overall market share of 64.1% followed by the Apple’s iOS based smartphones (18.8% market share). It is logical to conclude that with an ever increasing large penetration of smartphones, they are becoming favorite targets of intruders to launch malicious attacks [NDTV-Gadgets, 2012]. As expected,\(^1\)

\(^1\)among these mobile devices, 144.3 million are smartphones [Gartner, 2012].
Google’s Android platform is facing the major brunt of malware attacks due to its large market share. A sixfold increase in Android malware is reported for just one quarter (Jul 2012 - Sep 2012); as a consequence, more than 175,000 Android malware existed in Sep 2012 [Trend-Micro, 2012] – the majority of them belong to Trojans and Adware categories [Kindsight-Inc., 2012]. The important reason for this dramatic increase of Android mobile malware is its open-architecture for application distribution and this means that a user can obtain an Android application from any distributor and install it on its phone. The pirated versions of Android applications are available on a number of dubious online stores. Moreover, the official application store for Android – Google Play Store – does not enforce strict application review process. This policy of open architecture is totally different compared with iOS platform that requires a strict application review process, signing, runtime signature and integrity verification of smartphone applications. Since Android does not have these protection mechanisms; therefore, this provides a fertile nursery for malware writers to exploit vulnerabilities of OS to design, develop and distribute malware.

To counter the threat, a number of anti-malware companies — McAfee, Kaspersky, Avast, Norton, and Lookout [Smartphone-antivirus, 2011] etc. – have launched signature based malware detection solutions for Android platform. The zero-day (previously unknown) malware can employ simple evasion and obfuscation techniques to evade signature-based anti-malware products and static analysis based techniques [Sukwong et al., 2010] [Szor, 2005]. Consequently, researchers are focusing on dynamic runtime analysis and detection techniques for smartphones. But in addition to classical challenges – high detection accuracy, low false alarm rate, low detection delay, robustness against evasion attempts – a dynamic detection technique must be able to operate with limited resources – computing power, memory resources and battery utilization [Oberheide and Jahanian, 2010] – available on a smartphone.

To provide a solution to these mobile malware detection challenges, a novel, real-time malware detection framework – TStructDroid\(^1\) – is proposed that performs in-execution dynamic analysis of kernel Process Control Blocks (PCBs) on Android platform\(^2\). The framework applies information-theoretic analysis on logged time-series features, followed by segmentation and frequency component analysis of data to train a classifier to detect Android malware. The framework is tested on a real world dataset of 110 benign and 110 malware Android applications. The results of experiments indicate that the framework is able to detect unseen malware with more than 98% accuracy and less than 1% false alarm rate. The system degrades the performance of a low-end Android phone by 3.73% only with a detection delay of less than 100 msec.

The major contributions of our work are as follows:

- A formal framework that has the capability of extracting hidden information from time-series features set that has a large spread – making it very difficult to discriminate malware and benign processes;

- To define the concept of “cumulative variance” and show that once it is ap-

\(^1\)The feasibility of using process control blocks for malicious applications detection on Android is previously published as pilot study in [Akbar et al., 2013]

\(^2\)The feasibility of using process control blocks for malware detection on Linux has been previously established in [Shahzad et al., 2011b].
plied on the frequency transform (computed using *Discrete Cosine Transform (DCT)*) of time series features, it is able to detect and highlight small changes in features’ set of malware and benign and processes.

- To show that execution traces in Process Control Blocks (PCBs) of a running process can be used to detect malware in an effective and efficient manner.

- To prove “cumulative variance of frequency components obtained using DCT, over the time varying windows of PCB parameters of benign and malicious processes is linear and stable”; therefore, it (cumulative variance) can be reliably employed and utilized in malware detection.

- The *TStructDroid* framework – a realtime malware detection framework that uses in-execution dynamic analysis of kernel Process Control Blocks on Android platform – is presented that has high detection accuracy, low false alarm rate, small detection delay and processing overhead, and is robust against evasion attempts by “crafty attackers”.

- The dynamic detection framework is evaluated and tested on a relatively large real world dataset of 110 benign and 110 malware Android applications.

### 6.1.1 Organization of the Chapter

The rest of the chapter is organized as follows. The characteristics of benign and malware datasets are explained in Section 6.2. The emphasis is on describing carefully crafted datasets for training and testing of the proposed framework. Afterwards, in Section 6.3 the novel scheme that extracts hidden patterns in execution traces of malicious processes, by utilizing information in the kernel structures, is explained. In Section 6.4, the major components of proposed *TStructDroid* framework are presented and the working of each component is discussed in detail. The performance evaluation of the framework is presented in Section 6.5. The related work is briefly described in Section 6.6 with an emphasis on different direction of our work. Finally, we conclude the chapter with an outlook to the future work.

### 6.2 Android - Malware & Benign Datasets

The mobile malware dataset plays a critical role in evaluating and validating malware detection framework. We have collected real world malicious and benign Android applications to ensure that the detector is evaluated with real world applications. The extracted features from the process control blocks of running processes are logged in Android kernel and will be made available online. The malware detection framework is evaluated with a relatively large dataset (from the perspective of dynamic detection) consisting of 110 malware and 110 benign applications.

#### 6.2.1 Benign Dataset

The benign applications are chosen from the top featured applications at Google Play Store for Android applications. In order to ensure diversity, a good mix of different categories – Games, Image Viewers, Sketching tools, Text Editors, Image Editors,
Android Utilities such as Recorder, Dialer, Maps etc., and Misc. applications – is selected from the month of August 2012. The methodology adopted for selection is: (1) select the (famous) top downloaded applications that have a large user penetration; (2) select a good mix among different categories to ensure diversity, and (3) maintain balance between user-interactive (e.g. editor) and automated/background-service (e.g. download manager) applications.

### 6.2.2 Malware Dataset

Mobile malware on Android has exponentially increased in the last couple of years. We had to resort to Contagio Mobile Malware Mini Dump\(^1\) – a publicly available collection of Android malware because COTS AV companies were unwilling to share their datasets. The selected malicious applications belong to Trojans, Adware, Rootkits, Bots and Backdoors categories.

**Trojans** – a hidden trigger initiates the malicious activity in the background while the legitimate application executes at front – make up the majority of Android malware in our dataset. Some prominent trojan families\(^2\) included in our dataset are: N.Zimto, LuckyCat, Qicsomo, FakeTimer, SMSZombie, Loozfon, ZFT, Instagram, FakeToken, Ginimi, VDLoaded, CounterClank, Gamex, DroidDreamLight, FakeInstaller and Arspam Alsalah.

**Adware** category of malware consist of applications that try to monitor a user’s behavior – surfing patterns and online shopping interests etc. – or try to steal a user’s data – SMS messages, photos, location information, banking information etc. Some well known families are Plankton, DougaLeaker, Gonein60x, Steeks and FindAndCall etc. [Shahzad et al., 2013].

**Rootkits** get unrestricted access to a system by infecting its kernel. The typical use of rootkit exploits is: to hide malicious activity of other malware. Moreover, the rootkits do not run with limited permissions environment (like most other malware); therefore, they are able to monitor and modify other applications and critical resources on a smartphone device. Our dataset contains well known Android rootkits such as Z4Root:three, and ITFUNZ.supertools [Shahzad et al., 2013]. **Bots** receive commands from a server (bot master) and perform defined actions associated with them. A bot belongs to a large network of bots (known as botnet) and the complete army launches a distributed denial of service or spam attacks. The CI4.updater [Shahzad et al., 2013] bot is included in our dataset. **Backdoor** allows an attacker to create a hidden communication channel with the infected device. The challenge is that even if the exploited vulnerability is patched, the attacker can still access the device through the backdoor. DroidKungfu [Shahzad et al., 2013] is a well known example of backdoors in our dataset.

It is important to understand that these categories of malware are loosely defined and are not mutually exclusive (a malware application might belong to multiple categories at the same time). The uniqueness of our datasets compared with others is: a high percentage of Trojanized/Spyware Android applications. The key challenge is that typically they appear to behave like benign applications and the malicious activity covertly executes in a trojan code patched in a popular game. These covert

---


\(^2\)For interested readers, we provide the complete list of individual malicious applications of all malware families present in our dataset in [Shahzad et al., 2013].
activities are difficult to detect and hence create a stress scenario for evaluating the effectiveness of a malware detection framework. Our pilot studies prove that the raw information, contained in the process control blocks, is insufficient to discriminate between malware and benign applications; therefore, a novel information processing framework is proposed to extract the hidden patterns in the execution traces.

6.2.3 Creation of Training & Testing Datasets

After discussing the datasets, we now elaborate two test scenarios – real life and cross validation – to analyze the accuracy and effectiveness of the proposed detection framework.

6.2.3.1 Realtime Scenario

In real life scenario, the true challenge is that the framework should have the ability to detect zero-day (previously unknown) malware on the basis of malware model created during the training phase. In order to realize that scenario, we train the framework on 219 applications and then treat the remaining applications as unseen test applications; as a consequence, we create 220 different training folds. (Remember that we have 110 benign and 120 malware applications.)

6.2.3.2 Standard - Cross Validation Scenario

The 10-fold cross validation is the standard methodology, proposed in machine learning literature, for classification of datasets. Ten training and testing dataset combinations (folds) are created in such a way that the testing dataset consists of randomly chosen 10% of benign and 10% of malware applications; while the remaining applications are used in the corresponding training dataset.

From the perspective of anti-virus industry, real time scenario is more relevant but it can result in an over-fitting of the model during training (especially) if the results are skewed. To overcome this shortcoming of a real world scenario, we use the cross-validation strategy to verify that the relatively high detection rate of a framework is statistically significant and resilient to the changes in the dataset.

6.3 A Novel Scheme for Extracting Hidden Patterns from Execution Traces of Kernel Structures

The fundamental tenet of dynamic detection frameworks is: the execution pattern/behavior of benign and malicious processes differs significantly [Dini et al., 2012][Shahzad et al., 2011b]. To mine the difference in execution patterns, the framework monitors the change in time varying features of a process control block. In this section, we present the formal framework – the core of our detection framework – that enabled us to mine hidden patterns in the time series data.

We selected 99 preliminary parameters from the process control block (task_struct in Android kernel). Some interesting parameters are: number of page frames, volunteer and in-volunteer context switches, number of page faults,
virtual memory used, CPU time, number of page tables, file system resources, re-
resource counters of signal structure etc. During the execution of a process, the values
of these parameters are periodically logged. The time interval for logging (call it
time-resolution $\delta t$) has been set to 10 milliseconds\(^1\). The logged preliminary features
are processed in batches of fixed time intervals.

The scheme consists of five steps that extract hidden information from the logged
time series features: (1) analyze the extracted features, (2) filter relevant features
only, (3) remove redundant features, (4) create windows (batches) of fixed time
intervals, (5) apply different time series transforms, and (6) finally, calculate the
statistical features – capable of detecting mobile malware – from the transformed
dataset.

First, we analyze some important features that can be used to classify a process.
It is important to emphasize that even though the short listed features have the po-
tential for classification of a process, it is hidden in the raw time-series values that
significantly overlap for benign and malicious processes. Finally, a mathematical
framework is presented that takes the frequency transform of the time-series data,
segments the blocks and applies the novel concept of “extracting this hidden infor-
mation through transformation to frequency domain, segmenting into blocks, and
finally uses the novel concept of “cumulative variance” to mine hidden information.

6.3.1 Short-listing Time series Features

The first important step is to remove redundant features that are of little value in
detecting mobile malware. Let $X_{n,i}$ be the random variable that identifies time-
series values of a single feature $f_i$ extracted from the process control block of an
Android’s applications process $P_n$.

$$X_{n,t} = (x_{n,1}, x_{n,2}, \ldots, x_{n,T}) \forall x_{n,t}, 1 \leq t \leq T$$

so,

$$f_i = X_{n,t}, \ 1 \leq n \leq N_p$$

The logged preliminary features can be represented by a set $\mathcal{F}$:

$$\mathcal{F} = \forall f_i \mid 1 \leq i \leq N_f$$

Some of the preliminary features are “indexers”: they are used as identifiers
for indexing purpose only. Though they might be unique/distinct, they are mostly
assigned by the operating system and have no relationship with the behavior of a
process. Therefore, as a first step, such features $\mathcal{F}_{indexers}$ are identified and removed
from our features’ list $\mathcal{F}$.

$$\mathcal{F}_{indexers} = \forall f_i \in \mathcal{F}, f_i \text{ is an indexing feature}$$

Afterwards, statistical measures – time-series difference, mean and variance – of
the features are used to identify time varying features.

\(^{1}\)Shortly, the rationale of choosing this resolution will be explained.
Definition 1. **Time-series Difference of a feature.** The time-series difference of a feature $D_{f_i}(t)$ is defined as the absolute difference between the consecutive time-series values of a feature summed over all processes in a given time window:

$$D_{f_i}(t) = \sum_{n=1}^{N_p} (x_{n,i,t} - x_{n,i,t-1})$$

Definition 2. **Time-series Mean of a feature.** The time-series mean of a feature $M_{f_i}(t)$ is defined as the average of a feature's values summed over all processes in a given time window:

$$M_{f_i}(t) = \frac{1}{N_p} \sum_{n=1}^{N_p} x_{n,i,t}$$

Definition 3. **Time-series Variance of a feature.** The time-series variance of a feature $V_{f_i}(t)$ is defined as the average of squared deviation of a feature’s value from its expected value summed over all processes in a given time window.

$$V_{f_i}(t) = \sigma^2_{f_i}(t) = \frac{1}{N_p} \sum_{n=1}^{N_p} (x_{n,i,t} - \mu_{n,i,t})^2$$

We now define $\epsilon$ as a very small value that is negligible. A set of constant feature values – $F_{\text{constant}}$ – consists of the features having negligible time-series difference for all processes.

$$\forall f_i \in \mathcal{F}, \quad f_i \in F_{\text{constant}} \iff |D_{f_i}(t)| < \epsilon$$

A set of null feature values, $F_{\text{null}}$, is a set of features having time-series mean nearly equal to zero for all processes.

$$\forall f_i \in \mathcal{F}, \quad f_i \in F_{\text{null}} \iff |M_{f_i}(t)| < \epsilon$$

The mean and variance of a feature’s values determine its distribution. If the distribution of a feature in benign and malware processes is equal, it cannot be used to discriminate benign from malware applications; therefore, these types of features, $F_{\text{ident-dist}}$, can be removed from the feature’s set.

$$\forall f_i \in \mathcal{F}, f_i \in F_{\text{ident-dist}}$$

if and only if,

$$|M_{f_i}(t)_{\text{bengin}} - M_{f_i}(t)_{\text{malicious}}| < \epsilon$$

and

$$|V_{f_i}(t)_{\text{bengin}} - V_{f_i}(t)_{\text{malicious}}| < \epsilon$$

To further elaborate, let $p_{f_i}$ be the probability that a feature $f_i$ is used for classification, then:

$$p_{f_i} = \begin{cases} 
0 & \text{if } f_i \in F_{\text{indexers}} \\
0 & \text{if } f_i \in F_{\text{constant}} \\
0 & \text{if } f_i \in F_{\text{null}} \\
0 & \text{if } f_i \in F_{\text{ident-dist}} \\
1 & \text{Otherwise}
\end{cases}$$
Once we remove constant and irrelevant features from the features’ set by applying above-mentioned criteria, the remaining short listed features’ set, \( \mathcal{F}_{sel} \), (listed in Table 6.1) is given by:

\[
\forall f_i \in \mathcal{F}, \quad f_i \in \mathcal{F}_{sel} \iff p_{f_i} > 0
\]

![Graphs](image)

Figure 6.1: Time-series Mean of Some Preliminary Features for Benign and Malicious Processes

Figure 6.2: Time-series Variance of Some Preliminary Features for Benign and Malicious Processes

From the \( \mathcal{F}_{sel} \) set, the time-series mean and variance of some selected features are plotted in Figures 6.1 and 6.2 respectively to illustrate viability of our thesis: the information in process control block of a process can be used to discriminate benign applications from malware.

Now that we have shortlisted some important features containing potential classification information, we need to ask ourselves an important question: Can the mean values of the short-listed features be used without any further processing? If
Table 6.1: Short-listed task struct parameters ($F_{sel}$) of TstructDroid framework for classification purpose

<table>
<thead>
<tr>
<th>No.</th>
<th>Parameter(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>task.state</td>
<td>The current processing state of process</td>
</tr>
<tr>
<td>2</td>
<td>task.usage_counter</td>
<td>Task structure usage counter</td>
</tr>
<tr>
<td>3</td>
<td>task.prio</td>
<td>It holds dynamic priority of a process</td>
</tr>
<tr>
<td>4</td>
<td>task.static_prio</td>
<td>Static priority or nice value of a process</td>
</tr>
<tr>
<td>5</td>
<td>task.normal_prio</td>
<td>It holds expected priority of a process</td>
</tr>
<tr>
<td>6</td>
<td>task.policy</td>
<td>Scheduling policy of the process</td>
</tr>
<tr>
<td>7</td>
<td>task.active_mm$\rightarrow$mmmap$\rightarrow$vm_pgoff</td>
<td>The offset in vm file in page-size units</td>
</tr>
<tr>
<td>8</td>
<td>task.active_mm$\rightarrow$vm_truncate_count</td>
<td>Truncation count or restart address</td>
</tr>
<tr>
<td>9</td>
<td>task.active_mm$\rightarrow$task_size</td>
<td>The size of task virtual memory space</td>
</tr>
<tr>
<td>10</td>
<td>task.active_mm$\rightarrow$cached_holder_size</td>
<td>If non-zero, the largest hole below free-area-cache</td>
</tr>
<tr>
<td>11</td>
<td>task.active_mm$\rightarrow$free_area_cache</td>
<td>First hole of size cached-hole-size or larger</td>
</tr>
<tr>
<td>12</td>
<td>task.active_mm$\rightarrow$mm_users</td>
<td>Number of processes using this address space</td>
</tr>
<tr>
<td>13</td>
<td>task.active_mm$\rightarrow$map_count</td>
<td>Number of memory regions of a process</td>
</tr>
<tr>
<td>14</td>
<td>task.active_mm$\rightarrow$hiwater rss</td>
<td>Maximum number of page frames owned by the process</td>
</tr>
<tr>
<td>15</td>
<td>task.active_mm$\rightarrow$total_vm</td>
<td>Address space size of process (in terms of number of pages)</td>
</tr>
<tr>
<td>16</td>
<td>task.active_mm$\rightarrow$shared_vm</td>
<td>Number of pages in shared file memory mapping of process</td>
</tr>
<tr>
<td>17</td>
<td>task.active_mm$\rightarrow$exe_vm</td>
<td>Number of pages in executable memory mapping of process</td>
</tr>
<tr>
<td>18</td>
<td>task.active_mm$\rightarrow$reserved_vm</td>
<td>Reserved virtual memory for a process</td>
</tr>
<tr>
<td>19</td>
<td>task.active_mm$\rightarrow$nr_files</td>
<td>Number of page table entries of a process</td>
</tr>
<tr>
<td>20</td>
<td>task.active_mm$\rightarrow$end_data</td>
<td>Final address of data section (indicates the length of data section)</td>
</tr>
<tr>
<td>21</td>
<td>task.active_mm$\rightarrow$last_interval</td>
<td>Last fault stamp interval seen by this process</td>
</tr>
<tr>
<td>22</td>
<td>task.nvcsw</td>
<td>Number of involuntary context switches of a process</td>
</tr>
<tr>
<td>23</td>
<td>task.mcs</td>
<td>Number of voluntary context switches of a process</td>
</tr>
<tr>
<td>24</td>
<td>task.minflt</td>
<td>Minor page faults occurred for a process</td>
</tr>
<tr>
<td>25</td>
<td>task.majflt</td>
<td>Major page faults occurred for a process</td>
</tr>
<tr>
<td>26</td>
<td>task.fs_exd_counter</td>
<td>It holds file system exclusive resources</td>
</tr>
<tr>
<td>27</td>
<td>task.fs_lock</td>
<td>The read-write synchronization lock used for file system access</td>
</tr>
<tr>
<td>28-32</td>
<td>task$\rightarrow$signal$\rightarrow$utime, stime, gtime, ctime, mcs</td>
<td>Resource counters of signal structure for dead threads and child processes</td>
</tr>
</tbody>
</table>

the mean values of selected features have a different spread for benign and malicious applications, they can be used directly (or with little pre-processing) and given as input to a machine learning classifier. But this premise does not hold as is evident from Figure 6.3. On a similar note, closely compare Figures 6.3(a) to 6.1(a) and Figures 6.3(b) to 6.2(a) in juxtaposition. Even though the features have potential for classification, it is hidden in the raw time-series values which overlap significantly for benign and malicious processes. The reason for this overlap is: most of the mobile malware in our dataset act like benign applications for majority of their lifetime. The malicious activities happen during a short span of time and therefore go undetected if only the raw values of features are taken into account. The mathematical model that is capable of extracting the hidden information is now presented.

6.3.2 Redundant feature-instance elimination

The next step is: to filter redundant instances – representing the process state when it is not doing any useful work – of periodically logged features. For doing efficient classification, only distinct values of a feature are saved.

**Definition 4. Instance.** Let $x_{n,i,t_j}$ be the value of a feature $f_i \in F_{sel}$ for process $N_n$ at a time instance $t_j$, then an Instance $\mathcal{I}_{n}(t_j)$ consists of the set of values of selected
features stored for the process $N_n$ at the time instance $t_j$, then:

$$\mathcal{I}_n(t_j) = \{x_{n,i,t_j} \mid 1 \leq i \leq N_{f_{sel}}\}$$

**Definition 5.** *Instance Difference.* Let $x_{n,i,t_j}$ be the value of a feature $f_i \in \mathcal{F}_{sel}$ for a process $N_n$ at a time tick $t_j$, and $\mathcal{I}_n(t_j)$ be the corresponding instance set; while $\mathcal{I}_n(t_{j-1})$ be the instance set at the previous time tick $t_{j-1}$, then Instance Difference is defined as the maximum of absolute difference between corresponding feature values.

$$D_{inst}(\mathcal{I}_n(t_j)) = \max |x_{n,i,t_j} - x_{n,i,t_{j-1}}|$$

for $1 \leq i \leq N_{f_{sel}}$.

The redundant features’ values are removed by saving only one instance (among a set of consecutive instances) if the instance difference is zero. The selected instances $\mathcal{I}_{sel}$ are given by:

$$\mathcal{I}_n(t_j) \in \mathcal{I}_{sel} \iff D_{inst}(\mathcal{I}_n(t_j)) > 0$$

### 6.3.3 Time-series Segmentation and Frequency Information Extraction

The next step is to extract information from the remaining features to aid our classification process. First, the instances (consisting of time series data) are segmented into different blocks/windows. For each window, the frequency component information is extracted using a *Discrete Cosine Transform (DCT).*

For a given time-series window (segment) $k$ of size $T$ belonging to a process $N_n$, the frequency information $C_i(\omega)$ of feature $f_i \in \mathcal{F}_{sel}$ is given by:

$$C_i(\omega) = \alpha(\omega) \times \sum_{j=1}^{T} (x_{n,i,t_j}) \cos\left(\frac{\pi 2j + 1}{2T}\right)$$

where

$$\alpha(\omega) = \begin{cases} \sqrt{\frac{T}{2}} & \text{for } \omega = 0 \\ \sqrt{\frac{2}{T}} & \text{for } \omega \neq 0 \end{cases}$$
Each instance $I_j$ contains sets of frequency component information of all features.

$$I_j = \{ \mathcal{E}_i(\omega) \mid 1 \leq i \leq N_{f_{\text{sel}}} \}$$

The window $W_k$ is a set consisting of $\text{WinSize}$ such instances.

$$I_j \in W_k \mid \text{WinSize} \times (k - 1) < j \leq \text{WinSize} \times k$$

### 6.3.4 Variance Accumulation for Time-series Segments

Once the frequency components of each feature is computed, the change – a gradual process in time-series data instead of a sudden one – is modeled using the concept of **Cumulative Variance**. In cumulative variance, the variance of each segment is added to give a better estimation of the overall change in the frequency components of different time-series segments.

The variance $\sigma^2_{W_k}$ of a window $W_k$ is:

$$\sigma^2_{W_k} = \frac{1}{N} \sum_{i=1}^{N} (I_k - \mu_{W_k})^2,$$

where $\mu_{W_k}$ is the mean of the frequency component values for all instances in a window $W_k$.

**Definition 6. Cumulative Variance of a window.** The cumulative variance $g_k$ for each window $W_k$ is defined recursively as the sum of variance $\sigma^2_{W_k}$ of window $W_k$, and cumulative variance $g_{k-1}$ of previous window $W_{k-1}$:

$$g_k = \sigma^2_{W_k} + g_{k-1}$$

The initial value of cumulative variance is $g_0 = 0$.

In Figure 6.4, the time-series cumulative variance corresponding to some selected features for benign and malicious processes is depicted. The change in execution patterns of benign and malicious processes is succinctly reflected in these graphs. A machine learning classifier can learn these gradual changes to create malware detection model. In case of rule based classifiers, the classification rules can also be generated.

Now, we will present two important properties of the cumulative variance: (1) it converges after some time; and (2) it can be modeled with a stable linear autoregressive model.

Before we prove the first property, we show that the recursive definition of cumulative variance can be converted to an iterative equation. This result will be useful in proving the convergence property for our proposed model.

**Lemma 1.** Let $g_k$ be the cumulative variance of window $k$ of size $S$ where $1 \leq k \leq K$ and $k$ is defined as $K = T/S$, then:

$$g_k = \sum_{k=1}^{K} (\mathbb{E}[X^2_k] - \mathbb{E}^2[X_k])$$
Figure 6.4: Time-series Cumulative Variance for Benign and Malicious Processes

Proof. We have defined cumulative variance $g_k$ as

$$g_k = g_{k-1} + f(X_k)$$

where

$$f(X_k) = \sigma^2(X_k)$$

Expanding $g_{k-1}$ and iterating, we get

$$g_k = f(X_1) + f(X_2) + f(X_3) + \ldots + f(X_K)$$

Substituting $f(X_k) = \sigma^2(X_k)$,

$$g_k = \sigma^2(X_1) + \sigma^2(X_2) + \sigma^2(X_3) + \ldots + \sigma^2(X_K)$$

We know that:

$$\sigma^2(X_k) = \text{E}[(X_k - \mu)^2]$$

$$\sigma^2(X_k) = \text{Cov}(X_k, X_k)$$

$$\therefore \sigma^2(X_k) = \text{E}[X_k^2] - \text{E}^2[X_k]$$
Now, by combining Equation (6.1) and (6.2):

\[ g_k = \mathbb{E}[X_1^2] - \mathbb{E}^2[X_1] + \ldots + \mathbb{E}[X_K^2] - \mathbb{E}^2[X_K] \]  

(6.3)

Hence,

\[ g_k = \sum_{k=1}^{K} (\mathbb{E}[X_k^2] - \mathbb{E}^2[X_k]) \]  

(6.4)

Using the iterative equation for cumulative variance derived in Lemma 1, we now prove that our proposed model converges after a specific time, and thus its trend within a segment can be used reliably for differentiating between benign and malicious applications.

**Theorem 1.** The function \( \frac{g_k}{g_{k-1}} \) is given by

\[ \frac{g_k}{g_{k-1}} = 1 + \frac{\sigma^2(X_k)}{g_{k-1}} \]  

(6.5)

and it is a bounded function and converges to 1, as time \( T \to \infty \)

**Proof.** Using Lemma 1

\[
\frac{g_k}{g_{k-1}} = \frac{\sum_{k=1}^{K} (\mathbb{E}[X_k^2] - \mathbb{E}^2[X_k])}{\sum_{k=1}^{K-1} (\mathbb{E}[X_k^2] - \mathbb{E}^2[X_k])}
\]

\[
= \frac{\sum_{k=1}^{K} (\mathbb{E}[X_k^2]) - \sum_{k=1}^{K} (\mathbb{E}^2[X_k])}{\sum_{k=1}^{K-1} (\mathbb{E}[X_k^2]) - \sum_{k=1}^{K-1} (\mathbb{E}^2[X_k])}
\]

\[
= \frac{\sum_{k=1}^{K-1} (\mathbb{E}[X_k^2]) + \mathbb{E}[X_K^2]}{\sum_{k=1}^{K-1} (\mathbb{E}[X_k^2]) + \mathbb{E}^2[X_K]} - \frac{\sum_{k=1}^{K-1} (\mathbb{E}^2[X_k])}{\sum_{k=1}^{K-1} (\mathbb{E}[X_k^2]) + \mathbb{E}^2[X_K]}
\]

\[
= \frac{\mathbb{E}[X_K^2]}{\mathbb{E}^2[X_K]} + \frac{1}{1 - \frac{\sum_{k=1}^{K-1} \mathbb{E}[X_k^2]}{\sum_{k=1}^{K-1} (\mathbb{E}[X_k^2])}} - \frac{1}{1 - \frac{\sum_{k=1}^{K-1} \mathbb{E}^2[X_k]}{\sum_{k=1}^{K-1} (\mathbb{E}[X_k^2])}}
\]

When accumulated variance is large,

\[
\sum_{k=1}^{K} \mathbb{E}[X_k^2] \gg \sum_{k=1}^{K} \mathbb{E}^2[X_k]
\]

Therefore, as \( \lim_{g_{k-1} \to \infty} \)

\[
\frac{\sum_{k=1}^{K-1} \mathbb{E}^2[X_k]}{\sum_{k=1}^{K-1} \mathbb{E}[X_k^2]} \to 0, \quad \frac{\sum_{k=1}^{K-1} \mathbb{E}[X_k^2]}{\sum_{k=1}^{K-1} \mathbb{E}^2[X_k]} \to \infty
\]
6.3 A Novel Scheme for Extracting Hidden Patterns from Execution Traces of Kernel Structures

Figure 6.5: Convergence of Cumulative Variance Rate for task→usage.counter (Theorem 1)

and

$$\frac{\mathbb{E}[X_{K}^2]}{\sum_{k=1}^{K-1} (\mathbb{E}[X_k^2] - \mathbb{E}^2[X_k])}, \quad \frac{\mathbb{E}^2[X_K]}{\sum_{k=1}^{K-1} (\mathbb{E}[X_k^2] - \mathbb{E}^2[X_k])} \to 0$$

thus, by applying the limits to the above equation for \( \frac{g_k}{g_{k-1}} \) becomes

$$\lim_{g_{k-1} \to \infty} \frac{g_k}{g_{k-1}} = 1$$

Having proved the convergence property, we now show that the given model applied to our dataset can be modeled as a linear and stable process. This result determines the suitability of the application of this scheme on our dataset.

**Theorem 2.** The presented model – cumulative variance of frequency components obtained using DCT, over the time varying windows of PCB parameters of benign and malicious processes – is linear and stable.

**Proof.** Let \( AR(p) \) be an autoregressive model of order \( p \). We define \( g_k \) according to \( AR(p) \) model as follows:

$$g_k = c + \sum_{i=1}^{p} \varphi_i g_{k-i} + \epsilon_k$$

where \( \varphi_1 \ldots \varphi_p \) are the parameters of the model, \( p \) is empirically evaluated as 4 and \( c = 1 \). For the model to be stable, the roots of polynomial \( z^p - \sum_{i=1}^{p} \varphi_z z_{p-i} \) must lie within the unit circle.

\( AR \) parameters are calculated using method of moments (Yule walker equations)

$$\gamma_m = \sum_{k=1}^{p} \varphi_k \gamma_{m-k} + \sigma^2_{\epsilon_k} \delta$$

(6.6)
6. TStructDroid: Realtime Malware Detection using Time-series Analysis of PCB on Android

Figure 6.6: 3D Visual Representation of Autoregressive Models for Benign and Malware processes

Where, $m = 1, \ldots, p$, $\sigma^2_{\epsilon_k}$ is the standard deviation of the input noise $\epsilon_k$, $\gamma_m$ is the autocorrelation function of $g_k$. Equation (6.6) forms a system of equations that can be represented in matrix notation and solved for $\{\varphi_k; m = 1, 2, 3, \ldots, p\}$, once autocorrelation function $\gamma_m$ of $g_k$ is known. For $m = 0$, we solve separately using the following equation.

$$\gamma_0 = \sum_{k=1}^{p} \varphi_k \gamma_{-k} + \sigma^2_{\epsilon}$$

This can be solved for $\sigma^2_{\epsilon}$ once $\varphi_m$ are known. After evaluating the model fit in all processes, we find that the average model fit measure is $\approx 85\%$. This proves our initial hypothesis that the variance accumulation of frequency components obtained using DCT, over the time varying windows of PCB parameters of benign and malicious processes can be modeled by a linear, stable statistical model.

By visualizing the coefficients of statistical autoregressive (AR) model (see Figure 6.6), we can make sure that the underlying process model of benign and malicious applications is intrinsically different. With this, the presentation of the mathematical framework of TStructDroid is complete.

6.4 TStructDroid Framework

In this section, we present the architecture of TStructDroid framework for detection of malware applications on Android smartphones. This framework employs an in-execution dynamic analysis, based on cumulative variance analysis of time-series features (using the mathematical framework of the previous section) obtained from monitoring of kernel process control blocks\(^1\) on Android smartphones.

\(^1\)Process Control Blocks in Linux/Android kernel are typically referred as Task Structures (task_struct).
When a new application is launched on Android, the Binder Inter Process Communication (IPC) mechanism is used for sending a message to the Zygote process. The Zygote process is a special process that is basically an instance of Dalvik VM with core libraries loaded as read-only. The Zygote process forks a new process to launch a new application in a new Dalvik VM without unnecessary copying of shared core libraries [Ehringer, 2010]. As a result of the fork system call from Zygote, the kernel creates a child process and adds its process control block (task_struct) to a doubly linked circular linked list. The scheduler executes these processes – in a round robin fashion – by sharing the time of a processor to create an illusion of multitasking.

The TStructDroid framework runs in the kernel space as a (Loadable kernel module) LKM and root privileges are required to load and execute it. The kernel module is compiled and tested on Samsung Galaxy Young device with Android Gingerbread distribution (Android 2.3.6, Kernel version 2.6.35.7). The kernel module executes periodically and performs the cumulative variance based analysis of an executing process.

The framework consists of three components: (1) features logger, (2) features analyzer & processor, and (3) classification engine. The feature logger – implemented in the kernel module – periodically dumps the contents of the process control blocks of the running processes from the doubly linked list of task_structs. The features analyzer & processor component is responsible for short-listing the features after removing redundant instances, applying time-series DCT transforms on the short-listed features, and finally applying cumulative variance based formal framework of Section 6.3 to train decision tree classifier to build classification rules. The framework uses majority voting on a window (segment) of feature instances to make a decision about an executing application (benign or malware).

The architecture of the proposed framework is shown in Figure 6.7 and now we explain its architecture and functionality of each component in detail.

Figure 6.7: Block diagram of malicious applications detection on Android in realtime using process control blocks (task_struct) - process flow in user and kernel space
6.4.1 Features Logger

Recall from Section 6.3 that the framework has selected only 32 features from 99 preliminary parameters of process control block (task struct in Android kernel). The selected features are listed in Table 6.1. The features logger module periodically saves the preliminary features (in windows or batches) obtained from each process control block in the task struct circular linked list.

6.4.2 Features Analyzer

The features’ analyzer component is responsible for removing redundant instances, creating windows (batches of instances), and applying time-series DCT transforms. It also utilizes the cumulative variance analysis on the extracted features that helps in identifying hidden patterns.

6.4.3 Classification Engine

After doing cumulative variance analysis of frequency components of the current time-series segment (representing the execution behavior of the processes), the machine learning classifier is delegated the task to learn the benign and malware models and then make a decision about the category of an executing application.

In order to select a suitable classifier for the extracted features’ set, Information Gain (IG) and Information Gain Ratio (GR) of the extracted frequency components are calculated and plotted in Figure 6.8 (the plot only depicts selected features’ set.)

It is evident that selected features have high information gain and information gain ratio. Decision tree based classifiers use IG and GR for building and pruning decision tree models. Therefore, a decision tree based classifier such as J48 appears to be a suitable candidate for the proposed framework. Moreover, in [Witten and Frank, 2005], it is shown that J48 is resilient to class noise because it avoids over
facing during learning and also prunes a decision tree for an optimum performance (a characteristic desired in resource constrained smartphones).

6.4.4 Voting Method & Alarm

The classifier gives \textit{Benign} | \textit{Malicious} decision, after doing cumulative variance based analysis of the frequency components in each time-series segment. The majority voting method is used to make a final decision about the category of an application – if among $W_{vote}$ consecutive segments, more than half of segments are malicious, the application is classified as malware and is killed; otherwise, the process (and its dynamic analysis) is allowed to continue.

6.5 Performance Evaluation

Now we report the accuracy of the proposed framework tested on a dataset consisting of 110 benign and 110 malicious real-world Android applications.

6.5.1 Classification Performance

In a typical two-class problem, such as detecting a process as malicious or benign, the classification decision may fall into one of the following four categories: (1) true positive (TP), classification of a malicious process as malicious, (2) true negative (TN), classification of a benign process as benign, (3) false positive (FP), classification of a benign process as malicious, and (4) false negative (FN), classification of a malicious process as benign. The classification performance is measured using three standard parameters: (1) Detection Rate ($\text{DR} = \frac{TP}{TP+FN}$), (2) False Alarm Rate ($\text{FAR} = \frac{FP}{FP+TN}$), and (3) Detection Accuracy ($\text{DA} = \frac{TP+TN}{TP+TN+FP+FN}$).

As mentioned in Section 6.2, the framework is tested in two different scenarios (Real world scenario and Standard cross-validation scenario). In order to study the behavior of the framework, the time-resolution ($\delta t$) (after which the features are logged) and the size of each segment ($T$) is gradually varied. The time resolution $\delta t$ is varied from 10ms to 40ms, and segment size $T$ is varied between 5, 10, 20 and 40 instances of frequency components information. The classification results for real world scenario and cross-validation scenario are tabulated in Table 6.2.

6.5.1.1 Real-time Scenario

As mentioned in Section 6.2, 220 different training and testing datasets are generated for this scenario, each representing a case of zero-day (unseen) malware detection. The results in Table 6.2 indicate that the framework can detect unseen malware applications on Android with a detection rate of above 98% and a false alarm rate below 1% in a consistent manner. The overall detection accuracy lies within 98.6 – 99.5%. These results illustrate the power of cumulative variance analysis of frequency components of time-series features obtained from process control block structure of the applications in detecting malware while they are still executing.
### Table 6.2: Classification Performance Results for Real-time & Cross-validation scenarios

<table>
<thead>
<tr>
<th>Time Resolution</th>
<th>Segment Size</th>
<th>Voting Window Size</th>
<th>Real-Time Scenario (%)</th>
<th>Cross-validation Scenario (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>δt ms</td>
<td>T instances</td>
<td>W vote segments</td>
<td>DR</td>
<td>FAR</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>30</td>
<td>99.09</td>
<td>0.91</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>30</td>
<td>99.09</td>
<td>0.91</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>30</td>
<td>98.18</td>
<td>0.90</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>30</td>
<td>99.09</td>
<td>0.90</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>30</td>
<td>99.09</td>
<td>0.90</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>30</td>
<td>99.09</td>
<td>0.90</td>
</tr>
<tr>
<td>40</td>
<td>5</td>
<td>30</td>
<td>99.09</td>
<td>0.90</td>
</tr>
<tr>
<td>40</td>
<td>10</td>
<td>30</td>
<td>99.09</td>
<td>0.90</td>
</tr>
<tr>
<td>40</td>
<td>20</td>
<td>30</td>
<td>99.09</td>
<td>0.90</td>
</tr>
<tr>
<td>40</td>
<td>40</td>
<td>30</td>
<td>100</td>
<td>0.90</td>
</tr>
</tbody>
</table>
6.5 Performance Evaluation

6.5.1.2 Standard Cross-validation Scenario

As mentioned in Section 6.2, 10 different folds are created for training and testing. The results shown in Table 6.2 indicate that the framework is able to detect malware applications on Android with a detection accuracy of above 92% consistently. The detection rate lies within 90 – 93.6%. This means that the framework detected at least 9 out of every 10 randomly chosen malware application for which it was not trained. However, as expected, the false alarm rate is relatively higher and varies between 5.4% and 7.3%. It is, however, pertinent to mention that the proposed framework will never be subjected to cross-validation scenario.

We now analyze the effect of changing the values of different choice parameters of the framework. For each choice, first an intuition based expected outcome is presented that is subsequently validated through experimental results.

6.5.1.3 Effect of change in Time Resolution ($\delta t$)

As specified earlier, the term time resolution ($\delta t$) is used to specify the time interval after which values of parameters in the process control block structure of a process are logged. A small value of $\delta t$ means that the process will be monitored more frequently that enables the framework to detect minor variations of parameters – enabling the framework to detect processes that mostly behave as benign and carry malicious activity only for short interval of time. The disadvantage, however, is that this can also lead to higher false alarms because transient short changes (mostly) are not a good indicator of a malicious activity. In comparison, if $\delta t$ is large, it can significantly help in reducing the processing overhead of monitoring a process leading to reduced number of context switches and small memory for storing the time-series data. From Table 6.2, it is obvious that a change in $\delta t$ has little effect on the detection rate. For example, the detection rate stays 99.09% (Real word) and 90% when $\delta t$ is varied from 10ms to 40ms ($T = 5, W_{vote} = 30$). In comparison, FAR decreases from 7.27% to 6.36% (Cross-validation) when $\delta t$ is varied from 10ms to 40ms ($T = 40, W_{vote} = 30$). It is important to emphasize that the overhead of features’ logging decreases by a factor of 4 for $\delta t = 40$ms as compared to $\delta t = 10$ms.

6.5.1.4 Effect of change in Segment Size ($T$)

Recall that the time-series data is divided into segments of fixed length to do cumulative variance analysis. A relatively large size of segment would improve the detection accuracy at the cost of relatively large detection delay (the time to detect a malware). The results in Table 6.2 support these arguments; however, the improvement is marginal – from 99.1% to 99.55% for real world and from 92.27% to 93.64% for cross-validation when segment size $T$ is increased from 5 instances to 40 instances ($\delta t = 40$ms, $W_{vote} = 30$).

6.5.1.5 Effect of change in Voting Window Size ($W_{vote}$)

The classifier gives Benign or Malicious verdict in each time-series segment after applying the cumulative variance analysis. Recall that the majority voting in a window of segment is used to make the decision. Intuitively speaking, a larger voting window should increase the detection accuracy and decrease the false alarm rate. This, however, will come at a larger detection delay. Moreover, malicious
activities/applications that are active for a short duration might not be detected. The results in Table 6.2 are reported for a fixed value of $W_{vote} = 30$ segments. This value is empirically chosen to optimize performance in terms of DA and FAR (see Figure 6.9). The results for cross-validation scenario are similar to Figure 6.9 and are skipped for brevity.

To conclude, the framework is able to detect mobile malware with a high detection accuracy and relatively low false alarm rate.

### 6.5.2 Processing Overheads

If a malicious application detection framework were to be deployed on smartphones, it is imperative that its processing overheads be small; otherwise, the users would be tempted to turn off the security framework to have a responsive phone. In the subsequent subsections, the processing overheads of the framework are analyzed.

#### 6.5.2.1 Device under test

The experiments are performed on a Samsung Galaxy Young S5360 device with an 832 MHz ARMv6 processor, 290 MB RAM and 160 MB built-in storage. A low end device was selected for the experiments to benchmark the processing overheads of the framework on a limited memory and processing power device.

#### 6.5.2.2 Estimation of processing overhead

As a first step, the running time of different modules of the framework, executed in one classification cycle, is estimated. Assume a time-resolution of $\delta t = 10$ ms, a segment size $T = 10$ instances and voting window size $W_{vote} = 30$ segments. One classification cycle, as a result, would last $\delta t \times T \times W_{vote} = 3000$ ms.

The 32 features are logged every 10 ms by doing a context switch to the kernel space. The execution time for large number of iterations is computed and then averaged to get an estimated figure of 48.83 $\mu$s to log features. 30 $\mu$s are required...
6.5 Performance Evaluation

Table 6.3: System Performance Degradation Analysis on Android

<table>
<thead>
<tr>
<th>Application</th>
<th>Baseline Exec. Time (s)</th>
<th>Exec. Time with TstructDroid (s)</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zip (Archive)</td>
<td>127.2</td>
<td>131.5</td>
<td>3.27</td>
</tr>
<tr>
<td>Zip (Best Compress)</td>
<td>67.86</td>
<td>69.94</td>
<td>2.97</td>
</tr>
<tr>
<td>Zip (Best Speed)</td>
<td>50.39</td>
<td>52.01</td>
<td>3.11</td>
</tr>
<tr>
<td>Zip (Deflated)</td>
<td>132.12</td>
<td>141.81</td>
<td>6.83</td>
</tr>
<tr>
<td>Zip (Filtered)</td>
<td>147.59</td>
<td>154.27</td>
<td>4.33</td>
</tr>
<tr>
<td>File Copy</td>
<td>190.6</td>
<td>196.95</td>
<td>3.22</td>
</tr>
<tr>
<td>File Search</td>
<td>131.68</td>
<td>134.91</td>
<td>2.4</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td></td>
<td><strong>3.73</strong></td>
</tr>
</tbody>
</table>

to check whether the instance is redundant and then removing it. For $T = 10$ and $W_{vote} = 30$, the average overhead of $10 \times 30 \times (48.8284\mu s + 30\mu s) = 23.648ms$ is estimated for the complete classification cycle consisting of context switches, logging and eliminating redundant instances.

Calculating frequency components in a segment using Discrete Cosine Transform (DCT) takes $320\mu s$ on the average and for $10 \times 32$ matrix of feature values in one segment $(30 \times 320\mu s = 9.6ms$ per classification cycle. Similarly, calculating accumulated variance of the frequency components takes on average approximately $570.63\mu s$ per segment and it is $(30 \times 570.63\mu s = 17.119ms$ for classification cycle. The testing phase of J48 classifier introduces a delay of $5\mu s$ per segment and it comes out to be $(30 \times 5\mu s = 0.15ms$ per classification cycle. Finally, the voting process takes $0.03ms$ for making a decision and raising an alarm if required.

To conclude, in a classification cycle of $3000ms$, the combined overhead of all components of the framework (on average) is approximately $23.648ms + 9.6ms + 17.119ms + 0.15ms + 0.03ms = 50.547ms$ or $1.685\%$. Such a small overhead makes the proposed framework a suitable candidate for malware detection on smartphones.

6.5.2.3 System performance degradation

Generally on an Android phone, a number of system services are running in the background in addition to foreground applications. Moreover, different applications have different execution patterns – CPU intensive or I/O intensive. Another aspect is the memory requirement that might result in different frequency of page faults. Therefore, the true deterioration can be significantly higher compared with the above-mentioned estimate.

The experiments are performed to measure the performance degradation, experienced by an application, when the framework is fully operational. We have created an ensemble of customized Android applications that perform different operations. Some of them are CPU-intensive, some are I/O-bound, while others are a combination of both. The memory requirements for applications vary as well. These operations include five different algorithms for compressing files, a file (folder) to copy them and the text search within a file. Four different folders, containing different files (total folder size: 425MB, 500MB, 850MB and 1275MB) have been used as input to these applications. The average performance degradation is reported in Table 6.3. As expected, the performance overhead varies with the type of operation, but the average performance overhead is approximately (3.73%) that is higher than the 1.67% but still acceptable for real world deployment on a smartphone.
6.5.3 Evasion Analysis and Mitigation Techniques

In this section, we present that how a “crafty attacker” can attempt to write a malware that can evade the detection framework. Later, mitigation techniques are also suggested.

The first strategy that a malware writer can employ is: to mimic the execution behavior of a benign process. To do this, a malware must know the values of benign sets of features and their cumulative variance. However, the values of most features are tightly coupled with the configuration of a device (such as physical memory, cache, storage and paging mechanisms etc.), its typical use and other active processes and services executing in the kernel space. Therefore, it is reasonable to assume that estimating features’ set for a phone in a particular state will not be easy for a malware writer. Recall that in our dataset, a significant number of Trojan malware applications (applications that appear benign and masquerade as common useful applications) exist, which are effectively detected by the framework; therefore, the ability to hide malicious execution patterns inside benign patterns is not a useful evasion technique.

Rootkits present a significant threat because they infect a system’s kernel and thus get full access to the system. The typical use of rootkit exploits is to hide activity of a malware installed on the system. Moreover, the rootkits are not run in limited permissions environment like other types of malware and thus are able to monitor and modify other applications and sensitive resources on a smartphone. A rootkit can either monitor another benign process on the system and make the malware application mimic a similar behavior, or it can simply trick the framework by reading the process control block of another benign process when an attempt is made to read process control block of a malware. It can even suspend the functionality of the system. Luckily, the problem of rootkits is well known in the OS security community for many years. It is important to mention that if a rootkit is installed on a system, it means that the system is already compromised. It is possible to mitigate this threat by using the proposed framework in conjunction with any available rootkit detection frameworks for Android (and other smartphones) [Bickford et al., 2010] [Joy and John, 2011] [Brodbeck, 2012].

6.6 Related Work

Dynamic malware detection techniques intend to detect malicious programs during or after a program’s execution by leveraging the runtime information. Such techniques may involve monitoring execution patterns of programs, performing the taint-analysis and estimating their impacts on the OS. As a result, they are able to withstand code obfuscation or polymorphism techniques. A number of dynamic malware detection frameworks have been proposed in literature. In this section, we describe some of the latest proposed frameworks and their shortcomings.

Dini et al. have presented a multilevel anomaly detection technique for detecting Android malware (MADAM) [Dini et al., 2012]. The proposed framework operates in the kernel and the user space simultaneously and is capable of detecting previously unseen, zero-day malicious applications. A rich feature set is derived due to a multilevel view of the system events in both spaces. The K-nearest neighbors (KNN) algorithm is then applied during classification process. The operation of
the framework can be divided into training, learning and operation phases. The machine learning classifier can adapt to new changes by incorporating new feature vectors in training and learning set at run-time. An average detection rate of 93% along with an average false positive rate of 5% is reported for a very small dataset.

A hybrid framework for automatic malware detection Smartdroid is proposed in [Zheng et al., 2012] which monitors a user’s interaction with the interface. In the static analysis, a static path selector builds activity control graphs and function call graphs. Function call graphs are updated for indirect and event driven API calls by employing the technique given by Woodpacker [Zhou et al., 2012]. The dynamic analysis is performed by making a modification in the source code of Android framework while a restrictive component is added to limit the new activities that are created after interacting with the user interface. This framework is unsuitable for online analysis because it demands changes in the Android OS.

TaintDroid [Enck et al., 2010] is an information flow tracking tool for Android smartphones that gives dynamic taint tracking capability to the system. This framework can track multiple sources of private data by applying labels to the sensitive data. These labels work on four levels of tracking namely variable-level, method-level, file-level and message-level. An alarm is raised if the labeled data leaves the system through an un-trusted third party application. The authors have evaluated TaintDroid extensively over thirty most commonly used android applications and zero false positive is reported. The system overhead is large because labels are to be stored adjacent to sensitive data and then they need to be propagated. It has 14% processing overhead and puts 4.4% memory overhead. The major limitation of TaintDroid is that it only looks for explicit information flow; therefore, it is still possible to circumvent taint propagation through implicit flow of information.

In [Yan and Yin, 2012], the authors have proposed DroidScope which is a multi-level semantic analysis tool that performs dynamic profiling and information tracking to detect malicious behavior and privacy leaks in Android based smartphone applications. The tool runs in a virtual environment and logs instruction traces, API calls (at OS level and Dalvik VM level) and uses taint analysis to discover leakage of sensitive information. The tool has been tested on only two real world malware samples.

Android Application SandBox framework is proposed in [Blasing et al., 2010] for malicious software detection on Android. This framework first performs static analysis of user applications. The applications are then transmitted to a remote server that executes them in the sandbox. Then it performs clustering and does analysis of the generated logs to detect malicious patterns. The dataset used for testing Sandbox is small and analysis of the time and memory overhead is also not included. Another framework that uses a similar concept of decoupled security is an API based malware detection system Paranoid Android [Portokalidis et al., 2010].

Another dynamic analysis based framework called Crowdroid has been proposed which recognizes Trojan-like malware. It takes into account the fact that genuine and trojan affected applications differ in types and number of system calls during the execution of an action that requires a user’s interaction. The authors have reported results on a small dataset. The false-alarm rate is significantly high (20%). The authors have not discussed the robustness of their features set [Burguera et al., 2011].

Andromaly is another IDS which monitors both the system and user behavior by
observing several parameters, spanning from sensors activities to CPU usage [Shabtai et al., 2012]. The authors have developed four custom malicious applications to evaluate the ability to detect anomalies. They have created four different training/testing scenarios and reported good detection accuracy. Andromaly degrades performance of a smartphone by 10% with the malware detection time of 5 sec. Also, it is not evaluated over real world Android malware dataset.

Confused deputy attacks – deputing tasks to a more privileged application through publicly defined interfaces such as Intents – allow an application to escalate its privileges indirectly. The authors of [Dietz et al., 2011] have proposed Quire framework that attempts to solve this problem by restricting the inappropriate use of application’s permissions through its public interface and providing a trusted communication mechanism between applications using remote procedure calls.

Some of the common shortcomings of above-mentioned dynamic malware detection approaches are: (1) significant processing overheads, (2) evasion through mimicry attacks, (3) high false alarm rates, and (4) lack of testing on real-world malware dataset. Our framework has successfully overcome most of these shortcomings because it has relatively small overhead, small FAR, and is evaluated on a large dataset of real world Android applications.

The novelty of the framework is its “cumulative variance” based mathematical model that extracts hidden information in the kernel structures by applying the concept on DCT transform of the time-series raw features’ values. To the best of our knowledge, such a mathematical treatment of time-series data in process control blocks is not used (to date) in the literature.

6.7 Conclusion

The major contribution of this chapter is to show that “cumulative variance of frequency components obtained using DCT, over the time varying windows of PCB parameters of benign and malicious processes is linear and stable”; therefore, it (cumulative variance) can be reliably employed and utilized in malware detection. Using the formal treatment, TStructDroid framework is presented that is an efficient and effective malware detection tool (for Android smartphones) and it has the capability to analyze the runtime behavior of a mobile malware.

The proposed framework is lightweight with an average performance overhead of approximately (3.73\%) and it has the ability to detect a malware while it is still executing (in-execution malware detection). The framework is evaluated in “real world” and “cross validation” scenarios on a real world dataset consisting of 110 benign and 110 malware Android applications. The reported experiments show that the framework is able to achieve more than 98% accuracy and less than 1% false alarm rate. The framework can work with a number of existing solution to provide a robust solution against rootkits. These features make TStructDroid a suitable candidate for deployment on real world smartphones.
Chapter 7

Conclusions and Future Directions

7.1 Conclusions

A lightweight, hybrid, scalable and performance efficient malware detection framework is proposed based upon ELF Structure for static analysis and PCB runtime features to perform runtime dynamic analysis. The major objective of the dissertation has been to engineer and design a layered detection framework to detect malicious executables before execution and malicious processes while they are executing. The dissertation has utilized artificial intelligence and machine learning techniques to design malware detection techniques in user and kernel space of an operating system. The effectiveness of the proposed techniques is verified by empirical evaluation using real world malware on real smartphones.

Before, the summaries and contributions of different chapters are presented, it looks pertinent to recall the requirements and objectives mentioned in the first chapter of the dissertation: (1) light weight framework to detect zero-day malware, polymorphic variants of existing malware, (2) higher detection accuracy, (3) lower false alarm rate, (4) robustness of featureset against the evasion attempts of crafty attacker; (5) minimal processing overheads and detection delay, (6) security framework should reside on an end host with minimum performance degradation, and (7) proposed security solution should be evaluated using real world malware sample and real systems and devices etc.

To achieve the objective of zero-day malware detection – before their execution – in chapter 3, ELF-Miner is presented that is a prototypical framework on Linux to detect malicious executable by mining and statically analyzing the structural features extracted from different sections of ELF headers. To show the discrimination and classification potential of structural features of benign and malicious executable, a detailed forensic analysis is performed. The system achieves high detection accuracy with a negligible false positive rate and processing overhead. Moreover, scalability experiments are performed to demonstrate that ELF-Miner is able to provide high detection accuracy even with a small number of features. The robustness of framework against evasion attempts is shown by forging structural features; the results depict that the detection accuracy is not significantly degraded.

To achieve the objective of malicious processes’ detection at runtime, chapter 4 introduces a novel concept of genetic footprint of a process that consists of the process control blocks maintained inside the kernel. A comprehensive analysis is done
for parameters selection among a list of the extracted features. Moreover, keeping in view, the time-series characteristics, complexity and information gain of the dataset, suitable machine learning algorithms are selected for classification. The theory of using genetic footprint for malicious processes’ detection is empirically evaluated on real world malware (collected from online repository of offensive computing) for Linux. The results of experiments demonstrate that the framework achieves a high detection accuracy with a zero false alarm rate. The processing overhead and detection delay of the framework – to the best of our knowledge – are the smallest for an in-execution dynamic malware detection scheme. Finally, the resilience of the scheme against evasion attempts is also proven by experiments.

To achieve the core object of the dissertation, chapter 5 presents a hybrid framework to detect zero-day, polymorphic and repacked malware – while residing on a disk and during their execution – on Linux based smartphones. The framework consists of two components: (1) ELF structural tracer, and (2) PCB runtime tracer. The Linux malware are cross-compiled and evolved to demonstrate the proposed hybrid framework on ARM based Linux smartphones. Two different arrangements of dataset are presented: singleton and existere to demonstrate the scenarios of zero-day and polymorphic malware detection. The empirical evaluation exhibits that both the components detect polymorphic malware (results of existere dataset) successfully with a high detection accuracy and zero false alarm rate. Alternatively, both the components detect zero-day malware (results of singleton dataset) with high detection accuracy but lower than the polymorphic datasets. It is also discussed and argued – with references from literature – that the PCB runtime traces of packed and unpacked malware are similar; because all packed malware are unpacked before execution and during execution their PCB traces become similar as that of usual unpacked malware. The resilience of hybrid framework against structural and run-time evasion attempts is demonstrated by experiments. Moreover, malware detection delay and processing overhead of the framework is also measured and it is almost negligible as compared to similar overheads reported in literature for other security solutions. Last but not the least, this proposed framework is directly portable – without any modification or cross compilation – on recently launched Ubuntu Linux based superphone.

Chapter 6 accentuates the objective of the thesis and presents a realtime malware detection framework for Android platform – the most widespread smartphone OS constructed on Linux kernel – that performs dynamic analysis of smartphone applications and detects the malicious activities through in-execution monitoring of PCB in Android kernel to detect real-world malware. Malicious processes detection on Android is more challenging task, compared with other Linux based smartphones because majority of malicious applications available on this platforms are spyware or trojans, and malicious codes are embedded in usual and legitimate application available in app store. For this reason, our previous solution – used to detect malicious processes using runtime analysis of PCB on Linux – didn’t perform well on Android. Therefore, an algorithm is designed to analyze and utilize the underlying time-series information and distinct frequencies changes in PCB of benign and malicious processes. The algorithm consists of different steps and by using information theoretic analysis, time-series feature logging, segmentation and frequency component analysis of data, and a machine learning classifier, it is able to detect real world (zero-day) malware applications with high a detection accuracy on Android while
producing very low false alarms. Moreover, the detection delay and system performance degradation, caused by the framework, is very reasonable even for a low-end Android smartphone. All the above discussed performance metrics indicate that the framework is feasible for deployment on resource constrained Android devices.

7.2 Future Directions

Malicious applications on smartphones consist of a wide range of categories (trojans, spyware and rootkits etc.), types (packed or unpacked) and families (the functionality and threat code of a malware has a close resemblance to other malware considered as a family). The malware datasets presented in the dissertation, used to demonstrate security frameworks (in different chapters) cover a wide range of malware categories but all of them are unpacked malware. Our research work (presented in thesis) can be extended in different directions by following the instructions given below.

7.2.1 Structural Analysis to Detect Malicious Libraries (Native) on Android

Since, the malware detection framework based on static analysis of structural information of executables on Linux based smartphones is successfully demonstrated. Therefore, it is planned to detect Adware – a malware category on Android – using the structural information. The Adware use native libraries to perform their malicious activities. Therefore, it is planned to utilize static analysis based component on Android for the detection of malicious libraries (in native format) shipped with Android applications.

7.2.2 Empirical Evaluation for Packed malware detection using PCB Runtime Analysis

It is proven in chapter 5 (in principle) that the PCBs of packed and unpacked malware become similar (during execution) because all packed malware needed to be unpacked before their execution start. We plan to empirically evaluate the detection of packed malware using in-execution traces of PCB on smartphones – that support the execution of native code.

7.2.3 Empirical Evaluation for Packed malware detection using structural Analysis

The evaluation and presentation of malware detection using structural analysis is given in chapter 3 and chapter 5. In future, we plan to present a static and
structural analysis based framework to detect packed malware on Linux based smartphones. The reason is that most of the newly introduced malware are repacked or polymorphic variants of existing ones.

Conclusively, the future directions of dissertation are: (1) analyzing and detecting the malicious native libraries (in Adware) on Android, (2) detecting the packed malware executables on other Linux based smartphones using structural features, and (3) analyzing and detecting the packed malware using time-series PCB information on Linux based smartphones (i.e. superphones).
References


REFERENCES


Castro, M., M. Costa, and T. Harris (2006). Securing software by enforcing data-
flow integrity. In Proceedings of the 7th Symposium on Operating Systems Design
and Implementation, pp. 147–160.

Computer.

258–271. ACM.

communication in android. In International conference on Mobile systems, applications,
and services, pp. 239–252. ACM.

Software Engineering Notes 29(4), 34–44.

Christodorescu, M., S. Jha, S. Seshia, D. Song, and R. Bryant (2005). Semantics-
aware malware detection. In IEEE symposium on security and privacy, pp. 32–46.

tional Conference on Machine Learning, pp. 115–123. Morgan Kaufmann.

Sons.

airborne malware. Future Challenges in Security and Privacy for Academia and

Damopoulos, D., S. Menesidou, G. Kambourakis, M. Papadaki, N. Clarke, and
machine learning classifiers. Security and Communication Networks.


Davidson, D. and B. Livshits (2012). Morepriv: Mobile os support for application
personalization and privacy.

Davis, B., B. Sanders, A. Khodaverdian, and H. Chen (2012). I-arm-droid: A
rewriting framework for in-app reference monitors for android applications. IEEE
Mobile Security Technologies, San Francisco, CA.

Di Cerbo, F., A. Girardello, F. Michahelles, and S. Voronkova (2011). Detection of
malicious applications on android os. Computational Forensics, 138–149.

Dietz, M., S. Shekhar, Y. Pisetsky, A. Shu, and D. Wallach (2011). Quire:
lightweight provenance for smart phone operating systems. In USENIX Security
Symposium.


REFERENCES


REFERENCES


REFERENCES


PaX (2010). “GRSecurity, Complete documentation for the PaX project”. In "http://pax.grsecurity.net/docs/".
REFERENCES

PCMag. (2012). PC Magazine Encyclopedia. http://www.pcmag.com/encyclopedia_term/0,2542,t=virus+signature&i=53969,00.asp#fbid=T6SP5gICCaM.


Shahzad, F., M. Shahzad, and M. Farooq (2011b). In-execution dynamic malware analysis and detection by mining information in process control blocks of linux os. *Information Sciences*.


Superphone (2013). Ubuntu now fits your phone - superphone thats also a full pc. *Canonical-Ltd.*


