2021

Data Driven Analysis And Characterization Of Modern Android Malware

Qian Han
qian.han.gr@dartmouth.edu

Follow this and additional works at: https://digitalcommons.dartmouth.edu/dissertations

Part of the Computer Engineering Commons

Recommended Citation
Han, Qian, "Data Driven Analysis And Characterization Of Modern Android Malware" (2021). Dartmouth College Ph.D Dissertations. 74.
https://digitalcommons.dartmouth.edu/dissertations/74

This Thesis (Ph.D.) is brought to you for free and open access by the Theses and Dissertations at Dartmouth Digital Commons. It has been accepted for inclusion in Dartmouth College Ph.D Dissertations by an authorized administrator of Dartmouth Digital Commons. For more information, please contact dartmouthdigitalcommons@groups.dartmouth.edu.
DATA-DRIVEN ANALYSIS AND CHARACTERIZATION OF MODERN ANDROID MALWARE

A Thesis
Submitted to the Faculty
in partial fulfillment of the requirements for the
degree of
Doctor of Philosophy
in
Computer Science
by
Qian Han
DARTMOUTH COLLEGE
Hanover, New Hampshire
May 2021

Examining Committee:

V.S. Subrahmanian
V.S. Subrahmanian, Ph.D., Chair

Andrew T. Campbell Ph.D.

Soroush Vosoughi, Ph.D.

Salvador Mandujano, Ph.D.

F. Jon Kull, Ph.D.
Dean of the Guarini School of Graduate and Advanced Studies
Abstract

Google’s Android operating system was first announced to the public in 2007 and was installed on more than three billion mobile devices by 2019. With the prevalence of Android OS, Android malware has since proliferated. Android malware is malicious software designed to exploit Android operating systems running on smart devices. Some variants of Android malware have the capability of disabling the device, allowing a malicious actor to remotely control the device, track the user’s activity, lock the device, and so on. Moreover, the evolution and sophistication of modern Android malware obfuscation and detection bypassing methods have significantly improved in recent years, making many traditional malware detection methods (e.g. signature-based detection) obsolete. In the meantime, malware samples from the same family might disguise themselves with different functionality. These features might be relatively stable over time to keep their purpose, or evolve and change to cope with the emerged detection techniques. To tackle malware proliferation, we need a scalable Android malware detection approach that can easily and reliably identify malware applications. Although numerous malware detection tools have been developed, including system-level and network-level approaches, scaling up the effective and lightweight detection for a large package of apps remains challenging.

In this thesis, I propose data-driven methods to detect and analyze malicious Android applications through static, dynamic, and custom-designed advanced features. First, I propose the advanced features to improve both Android malware detection
results and the robustness of the detection system, specifically focusing on Android spyware, banking Trojans, and rooting malware. Second, I explore stability analysis in 122 general Android malware families and 120 Android goodware families through the definition of tau-Homogeneous Partition and the stability score of feature vectors in a period, which reveals the top stable and unstable features during general Android malware’s evolution over time. This thesis will help academics obtain a detailed picture of Android malware detection based on data-driven analysis using basic and advanced features. It will then act as a framework for subsequent studies to initiate new studies and continue to direct research in the area more broadly.
Acknowledgments

First, I thank my advisor Prof. V.S. Subrahmanian for the constant encouragement of my Ph.D. study and research. He has greatly shaped my understanding of research and taught me the value of producing high-quality research that reaches fields beyond computer science. He provided an encouraging atmosphere where I can interact and work with other researchers. I respect his truthful opinions, patience, and empathy throughout the past four years. I could not finish this study without his invaluable mentorship, advice, and encouragement.

Then, I felt lucky and honored to have the chance to work closely with Dr. Salvador Mandujano and Sebastian Porst, and Dr. Sai Deep Tetali from Google, Prof. Fabio Pierazzi from King’s College London, Prof. Ghita Mezzour from the International University of Rabat, Morocco in the Android malware-related projects. I enjoyed collaborating with Prof. Yuval Elovici and Prof. Rami Puzis for my internship at Telekom Innovation Laboratories at Ben Gurion University and Dr. Hao Zhong’s mentorship for my internship at Facebook. I would like to thank all present and former members from the Dartmouth Security and Artificial Intelligence Group: Yanhai Xiong, Haipeng Chen, Chiara Pulice, Chongyang Bai, Rui Liu, Dongkai Chen, Saksham Arora, Jack Sadoff, and so on. I am thankful for all the time they spent with me.

Finally, I am grateful to have the support from my family. I especially thank my fiancée Yumeng Gao for her affection, patience, and encouragement. Moreover, I can not thank my parents sufficiently. They provided me a decent education and are very
supportive of my whole Ph.D. life.
CURRICULUM VITAE

Qian Han

EDUCATION
Ph.D. Candidate in Computer Science, Department of Computer Science, Dartmouth College, United States, 2016.08 - 2021.05.
B.Sc. in Electronic Engineering, Department of Electronic Engineering, Tsinghua University, China, 2012.08 - 2016.07.

PUBLICATIONS


PROFESSIONAL EXPERIENCES

Research Scientist, Facebook, Inc. Starting from 2021.06.

Software Engineer Intern, Facebook, Inc. 2020.06 - 2020.09.

Research Intern, Deutsche Telekom Innovation Laboratories, advised by Prof. Yuval Elovici. 2019.06 - 2019.09.
Contents

1 Introduction 1
   1.1 Background ........................................ 1
   1.2 Contributions ..................................... 3
   1.3 Summary ........................................... 6

2 Malware Categories 7
   2.1 Distributed Denial of service .................... 8
   2.2 Backdoor ........................................... 9
   2.3 Spyware ........................................... 10
   2.4 Hostile downloader .................................. 11
   2.5 Privilege escalation ............................... 11
   2.6 Ransomware ........................................ 12
   2.7 Rooting ............................................. 13
   2.8 Trojan ............................................. 13
   2.9 SMS fraud .......................................... 14
   2.10 Spam .............................................. 14
   2.11 Toll fraud ......................................... 15

3 Static and Dynamic Features of Android Applications 16
   3.1 Static Features ..................................... 17
   3.2 Dynamic Features .................................. 23
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Advanced Features of Android Applications</td>
<td>30</td>
</tr>
<tr>
<td>4.1</td>
<td>API Features</td>
<td>30</td>
</tr>
<tr>
<td>4.2</td>
<td>Triadic Suspicion Graph based Features</td>
<td>32</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Triadic Suspicion Graph</td>
<td>32</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Suspicion Score</td>
<td>35</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Suspicion Rank</td>
<td>37</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Window-based TSG Feature Creator</td>
<td>39</td>
</tr>
<tr>
<td>4.3</td>
<td>Landmark-based Features</td>
<td>42</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Landmark Selection</td>
<td>43</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Landmark-based Feature Generation</td>
<td>45</td>
</tr>
<tr>
<td>4.4</td>
<td>Feature clustering-based Features</td>
<td>46</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Clustering and Feature Aggregation</td>
<td>48</td>
</tr>
<tr>
<td>4.5</td>
<td>Correlation Graph based Feature Transformation</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>Detection and Analysis of Android Malware Families</td>
<td>53</td>
</tr>
<tr>
<td>5.1</td>
<td>DBank: Predictive Behavioral Analysis of Recent Android Banking Trojans</td>
<td>53</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Detection Performance</td>
<td>54</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Key Features in Android Banking Trojan’s Detection</td>
<td>57</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Robustness Analysis</td>
<td>59</td>
</tr>
<tr>
<td>5.2</td>
<td>A Data-Driven Characterization of Modern Android Spyware</td>
<td>67</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Supervised Learning Algorithms and Ensemble Late Fusion (ELF)</td>
<td>68</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Evaluation</td>
<td>71</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Distinguishing Characteristics of Spyware</td>
<td>75</td>
</tr>
<tr>
<td>5.3</td>
<td>Android Malware Detection via (Somewhat) Robust Irreversible Feature Transformations</td>
<td>78</td>
</tr>
<tr>
<td>5.3.1</td>
<td>The FARM Dataset</td>
<td>78</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Experimental Evaluation</td>
<td>78</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Screenshot of the manifest file of the malware sample \textit{Zsone_245}</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>Permissions requested from the malware sample \textit{Alsalah_cf_rm_renaming}</td>
<td>20</td>
</tr>
<tr>
<td>3.3</td>
<td>The manifest of a trojan malware sample \textit{test.txt} showing receivers, services etc.</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>Screenshot of the API information of the malware sample \textit{Zsone_245}</td>
<td>23</td>
</tr>
<tr>
<td>3.5</td>
<td>Part of the source code of the malware sample \textit{Zsone_245} containing a url</td>
<td>24</td>
</tr>
<tr>
<td>3.6</td>
<td>Services started by \textit{Plankton} in Droidbox</td>
<td>25</td>
</tr>
<tr>
<td>3.7</td>
<td>Dexclass called by \textit{Plankton} in Droidbox</td>
<td>26</td>
</tr>
<tr>
<td>3.8</td>
<td>Permission required by \textit{Plankton} in Droidbox</td>
<td>26</td>
</tr>
<tr>
<td>3.9</td>
<td>Dataleaks of \textit{Plankton} in Droidbox</td>
<td>27</td>
</tr>
<tr>
<td>3.10</td>
<td>Recvnet operation by \textit{Plankton} in Droidbox</td>
<td>27</td>
</tr>
<tr>
<td>3.11</td>
<td>Sendnet operation by \textit{Plankton} in Droidbox</td>
<td>28</td>
</tr>
<tr>
<td>3.12</td>
<td>Receive actions by \textit{Plankton} in Droidbox</td>
<td>28</td>
</tr>
<tr>
<td>3.13</td>
<td>One of the file read and write operations by \textit{Plankton} in Droidbox</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>Illustration of TSG</td>
<td>34</td>
</tr>
<tr>
<td>4.2</td>
<td>Window-based API package ranking by descending SUS/SR scores</td>
<td>40</td>
</tr>
<tr>
<td>4.3</td>
<td>Illustration of landmark-based features with 6 applications, 2 landmarks and the Euclidean distance function</td>
<td>47</td>
</tr>
</tbody>
</table>
4.4 Illustration of FC features with $G = 2$ groups for 4—dimensional basic API features and averaging feature aggregation. ........................................ 50

4.5 Illustration of CG features with 2 communities detected and averaging feature aggregation method. .......................................................... 52

5.1 The training process setup ................................................................. 55

5.2 ABT vs Goodware: Histograms of distinguishing features values. .................. 58

5.3 ABT vs Other-malware: Histograms of distinguishing features values. .................. 58

5.4 Adversary Error Rate vs. Overlap Ratio divided by baseline: Manifest + Dynamic. Higher values imply better robustness. .......................... 61

5.5 Euclidean distance among attacker’s and defender’s feature sets centroids. High values imply higher robustness. ........................................... 63

5.6 Kolmogorov-Smirnov distance among attacker’s and defender’s feature sets centroids. High values imply higher robustness. ........................................... 64

5.7 AUC performance and adversary error rate of DBank’s TSG vs. traditional features using different proportions $\rho$ of the training set. ................. 66

5.8 Impact of Fake Call Attack on DBank’s Performance ..................................... 67

5.9 Scheme of the ELF architecture. .......................................................... 70

5.10 Impact of fake API package call attack on Android rooting malware detection: Goodware vs. Rooting Malware (No-Isomorphic) .................... 89

5.11 Impact of fake API package call attack on Android rooting malware detection: Other Malware vs. Rooting Malware (No-Isomorphic) ............. 89

5.12 Impact of increased percentage of permissions attack on Android rooting malware detection: Goodware vs. Rooting Malware (No-Isomorphic) .... 89

5.13 Impact of increased percentage of permissions attack on Android rooting malware detection: Other Malware vs. Rooting Malware (No-Isomorphic) .... 89
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.14</td>
<td>Impact of reduced percentage of API package call attack on Android rooting malware detection: Goodware vs. Rooting Malware (No-Isomorphic)</td>
<td>89</td>
</tr>
<tr>
<td>5.15</td>
<td>Impact of reduced percentage of API package call attack on Android rooting malware detection: Other Malware vs. Rooting Malware (No-Isomorphic)</td>
<td>89</td>
</tr>
<tr>
<td>5.16</td>
<td>Impact of fake API package call attack on Android Banking Trojans detection: Goodware vs. Banking Trojans (Isomorphic)</td>
<td>90</td>
</tr>
<tr>
<td>5.17</td>
<td>Impact of fake API package call attack on Android Banking Trojans detection: Other Malware vs. Banking Trojans (Isomorphic)</td>
<td>90</td>
</tr>
<tr>
<td>5.18</td>
<td>Impact of increased percentage of permissions attack on Android Banking Trojans detection: Goodware vs. Banking Trojans (Isomorphic)</td>
<td>90</td>
</tr>
<tr>
<td>5.19</td>
<td>Impact of increased percentage of permissions attack on Android Banking Trojans detection: Other Malware vs. Banking Trojans (Isomorphic)</td>
<td>90</td>
</tr>
<tr>
<td>5.20</td>
<td>Impact of reduced percentage of API package call attack on Android Banking Trojans detection: Goodware vs. Banking Trojans (Isomorphic)</td>
<td>90</td>
</tr>
<tr>
<td>5.21</td>
<td>Impact of reduced percentage of API package call attack on Android Banking Trojans detection: Other Malware vs. Banking Trojans (Isomorphic)</td>
<td>90</td>
</tr>
<tr>
<td>5.22</td>
<td>Impact of fake API package call attack on Android Spyware detection: Goodware vs. Spyware (No-Isomorphic)</td>
<td>91</td>
</tr>
<tr>
<td>5.23</td>
<td>Impact of fake API package call attack on Android Spyware detection: Other Malware vs. Spyware (No-Isomorphic)</td>
<td>91</td>
</tr>
<tr>
<td>5.24</td>
<td>Impact of increased percentage of permissions attack on Android Spyware detection: Goodware vs. Spyware (No-Isomorphic)</td>
<td>91</td>
</tr>
<tr>
<td>5.25</td>
<td>Impact of increased percentage of permissions attack on Android Spyware detection: Other Malware vs. Spyware (No-Isomorphic)</td>
<td>91</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.26</td>
<td>Impact of reduced percentage of API package call attack on Android</td>
<td>91</td>
</tr>
<tr>
<td></td>
<td>Spyware detection: Goodware vs. Spyware (No-Isomorphic)</td>
<td></td>
</tr>
<tr>
<td>5.27</td>
<td>Impact of reduced percentage of API package call attack on Android</td>
<td>91</td>
</tr>
<tr>
<td></td>
<td>Spyware detection: Other Malware vs. Spyware (No-Isomorphic)</td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>Example signal</td>
<td>94</td>
</tr>
<tr>
<td>6.2</td>
<td>Example signal. Assume the second value $x_2$ of the original signal in Figure 6.1 is influenced by the noise and increases to a value greater than $x_3$.</td>
<td>100</td>
</tr>
<tr>
<td>6.3</td>
<td>Demo GUI of Airpush Top stable feature signals</td>
<td>103</td>
</tr>
<tr>
<td>6.4</td>
<td>Demo GUI of Svpeng Top unstable feature signals</td>
<td>104</td>
</tr>
<tr>
<td>6.5</td>
<td>Distribution of average stability score on API Package features</td>
<td>105</td>
</tr>
<tr>
<td>6.6</td>
<td>Distribution of average stability score on Permission features</td>
<td>106</td>
</tr>
<tr>
<td>6.7</td>
<td>Distribution of average stability score on Opcode features</td>
<td>106</td>
</tr>
<tr>
<td>6.8</td>
<td>Distribution of average stability score on System Command features</td>
<td>107</td>
</tr>
<tr>
<td>6.9</td>
<td>$MG_{avg}$ with average stability score</td>
<td>108</td>
</tr>
<tr>
<td>6.10</td>
<td>$MG_{ent}$ with the entropy of stability score</td>
<td>108</td>
</tr>
<tr>
<td>6.11</td>
<td>$MG_{std}$ with the standard deviation of stability score</td>
<td>109</td>
</tr>
<tr>
<td>6.12</td>
<td>$MG_{avg}$ with mean stability score among API Package features</td>
<td>110</td>
</tr>
<tr>
<td>6.13</td>
<td>$MG_{ent}$ with the entropy of the stability score among Permission features</td>
<td>111</td>
</tr>
<tr>
<td>6.14</td>
<td>$MG_{std}$ with the standard deviation of stability score among Opcode features</td>
<td>111</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Dataset Composition .................................. 54

5.2 Multiple metrics (AUC and F1 etc.) on Android Banking Trojans detection vs. Goodware No-Isomorphic dataset. Best Late Fusion Parameters: API + S + D: 0.344, TSG: 0.116, LM: 0.105, FC: 0, CG: 0.435. ....................................................... 55

5.3 Multiple metrics (AUC and F1 etc.) on Android Banking Trojans detection vs. Other-malware No-Isomorphic dataset. Best Late Fusion Parameters: API + S + D: 0.345, TSG: 0.338, LM: 0.076, FC: 0, CG: 0.241. ....................................................... 56

5.4 Performance of classifiers in spyware vs goodware (average with 10-fold cross-validation). The best traditional classifier is RF. The best overall classifier is ELF with weights $w_{RF} = 0.711$, $w_{Wide&Deep} = 0.222$, $w_{BRBM} = 0.045$, and $w_{SVM} = 0.022$. The p-values are computed via a t-test comparing $F_1$-Scores. We also use standard statistical notation ***, **, * (**<0.01; **<0.05, *<0.1). ....................................................... 72
5.5 Performance of classifiers in **spyware vs other-malware** (average with 10-fold cross-validation). The best traditional classifier is RF, which is outperformed by ELF with positive weights $w_{RF} = 0.967$ and $w_{BRBM} = 0.033$. The p-values are computed through the t-test performed against the $F_1$-Score results and, in addition to the raw p-values, we also use the statistical confidence notation ***, **, * (***<0.01; **<0.05; *<0.1).
Chapter 1

Introduction

Section 1.1

Background

According to Statista [1], Android was the most widely used smartphone OS in the world with a market share of 87.9% at the end of the second quarter of 2017. Another Statista report [2] shows that Android had more than 3 times as many vulnerabilities reported in it as the nearest competing mobile operating system, iOS. Around 84.8% of smartphones sold globally use the Android operating system in 2020 [3]. At the end of Feb 2021, there were more than 3 million applications on Google Play, which is the official platform for Android applications [4]. Due to numerous reasons, such as the transparent ecological mode of Android apps, its coarse-grained permission control, and the potential to invoke third-party code, several security attack surfaces are present, which seriously undermines the reputation of Android applications. Statistics reveal that in 2016 alone, around 3.25 million Android applications that were compromised with malware were detected, which indicates that a fresh Android malware app was identified about every 10 seconds [5]. To maintain the stability of the Android environment, a range of strategies have
been suggested, including device reinforcement, vulnerability monitoring, developer reviews, and malware detection. Among the different defense solutions, Android malware detection is a commonly available security prevention mechanism that can avoid malware from being released into the Android device marketplace or being activated and used. Based on previous studies, Android malware detection and analysis technologies can be categorized into three categories: static analysis-based detection, dynamic analysis-based detection, and hybrid analysis-based detection. Static analysis is based on the analysis of suspicious code without running the Android script. It may achieve high data coverage but faces multiple countermeasures such as code obfuscation and complex code loading. Conversely, dynamic analysis requires the study of the Android application by running the code. This will reveal threats that are not possible to find through static analysis, but the computing resources and time expense of dynamic analysis are comparatively high. Hybrid analysis is a process that blends static analysis and dynamic analysis to strike a compromise between detection efficacy and performance. Besides, machine learning theory is commonly applicable in the identification of Android malware, whether based on static, dynamic, or hybrid analysis approaches. Compared with conventional approaches, such as signature-based malware detection, which is based on detecting unique characteristics in recognized malware, machine learning-based detection has the potential to recognize previously unknown forms of malware and may have better results in detection effectiveness and quality. Some previous research has addressed Android malware detection approaches relying on machine learning. However, there are some shortcomings in the previous research, that is most of the previous papers mostly used machine learning or deep learning approaches to identify and evaluate harmful Android apps and good applications, but did not concentrate on particular malware families, such as Android banking Trojans, Spyware, Rooting malware and
so on.

Section 1.2 Contributions

This thesis provides a comprehensive analysis of research within this particular field of work thus suggesting several innovative approaches to address these limitations. The key contributions of this thesis are described as follows:

(a) Given an Android APK, we automatically predict whether it is a banking trojan, spyware, rooting malware or not through the proposed android malware detection systems. We show that our detection systems achieve this with high accuracy, even after removing isomorphic feature vectors\(^2\).

(b) We propose a novel structure called a *Triadic Suspicion Graph* (TSG for short), along with two novel graph metrics called suspicion scores (SUS) and suspicion ranks (SR) that are derived from TSGs. Moreover, we present a novel Window-Based TSG Feature Creator. We show that TSG-based features alone, when used in conjunction with off the shelf machine learning algorithms, generate high predictive accuracy — but when used in conjunction with additional features derived from more traditional static and dynamic analysis \[35, 36, 37, 38, 39\] generate even better results. TSG-based features have some interesting defensive properties in the presence of adversaries who might guess and obtain even a large part of our training set. In the real world, attackers may subscribe to

\(^2\)We say that two APKs (hashes) are *isomorphic* when they have identical feature vectors. In such cases, cross-validation by splitting the data may cause both the training and validation sets in a given cross-validation fold to contain the same feature vectors, leading to an artificial and incorrect increase in all measures of predictive accuracy. Past efforts in using machine learning in cybersecurity do not say anything about the occurrence of isomorphic samples. In this paper, we present results after removing isomorphic samples, though a removable appendix does present the results on isomorphic samples. Note that an attacker can easily generate different hashes of the same samples by first decompiling the APK, then changing only the package names or file paths in the manifest, and finally repackaging it to generate a new APK with a different hash.
or have access to malware datasets (e.g., through VirusTotal). We show that even if the attackers have access to over 90% of the samples that we use, their classification accuracy will still be low. This is shown via multiple distance-based metrics as well as via a detailed Kolmogorov-Smirnov test. We additionally show that by a judicious choice of our training set from the set of openly available samples, we further compromise the ability of an adversary to reverse engineer our predictors.

(c) We conduct a thorough analysis of the features that best distinguish Android Banking Trojans from both goodware and other types of malware (e.g., ransomware, SMS fraud). In particular, we show that the following features play an important role: (i) requesting permissions to receive/modify SMS, read phone state, and control system alert windows are each highly indicative of ABTs, (ii) a low frequency of calls to some particular Android API packages (e.g., android.widget and android.view), (iii) and possible repacking activities through read, write and dynamic class load operations.

(d) We consider the task of separating spyware vs. goodware and spyware vs. other malware, and compare the performance of several traditional and deep learning methods. We show for the first time that ensemble learning with different classifiers leads to better 10-fold performance for identifying spyware. To the best of our knowledge, past ML-based malware detection studies have not studied spyware in depth—they mostly consider the general problem of distinguishing goodware vs. malware, and rely on just one individual classifier (e.g., SVM in [36], RF and variants in [9, 40]). Instead, we propose an ensemble late fusion (ELF) architecture that outperforms all individual classifiers.

(e) We identify the features that are key to separating modern Android spyware (between July 2016 and July 2017) from goodware and from other malware.
Existing work on malware characterization has not focused on spyware — in addition to malware vs. goodware classification, they look at related problems such as predicting malware spread [41] or distribution of goodware and malware in different Android markets [42].

(f) We propose FARM (short for Feature transformation based Android Malware detector), a framework for detecting Android Rooting Malware which is robust to certain types of attacks that we might expect malicious hackers to try out. FARM takes well-known features for Android malware detection and introduces three new types of feature transformations (Landmark based transformations, Feature clustering-based transformations, and Correlation graph-based transformations) that transform these features irreversibly into a new feature domain. We first test FARM on 6 Android classification problems separating goodware and “other malware” from 3 classes of malware: rooting malware, spyware, and banking trojans. We show that FARM beats standard baselines when no attacks occur. Though we cannot guess all possible attacks that an adversary might use, we propose three realistic attacks on FARM and show that FARM is very robust to these attacks in all classification problems.

(g) We present SAAM (short for Stability Analysis of Android Malware Families), which investigates how malware samples from the same family changing over time and the efficiency of detection techniques at the feature level. We also define optimal-partition, stability score, and three different kinds of MG (malware over goodware) ratio on the features over time in Android apk families, with a detailed stability score analysis on 122 families of malware between 2012-2019 from VirusTotal, and 120 families of goodware between 2012-2020 from Google Play Top free apps, with an average of 60 samples for each family. In the SAAM project, we also summarize 4 kinds of top stable and unstable features over
all the collected Android apk families, specifically on Android API features, Android Permission features, Operation Code features, and System Command features.

Section 1.3

Summary

In this thesis, we first introduce a novel family of structures called Triadic Suspicion Graphs (TSG) resilient to adversary attacks and achieve high accuracy even while only using a subset of training data. Then, we present the Ensemble Late Fusion (ELF) architecture that combines the results of multiple classifiers’ predicted probabilities to generate a final prediction of whether an Android app is spyware or not, and we also conduct a detailed analysis of some important features in the spyware families. Third, we propose FARM, which includes three new feature transformation techniques that can be used to generate feature vectors that are very hard to reverse engineer. Last, we introduce SAAM (Stability Analysis of Android Malware Families) framework and study how malware samples from the same family evolve over time, as well as the top stable or unstable features at the 4 different feature levels.

There also some limitations of this thesis work. First, we perform dynamic analysis through the Koodous online service, but it does not cover all possible execution traces. For the classification system based on machine learning, the cost of getting dynamic features is relatively high, and it is not easy to expand on a large scale. Then, we observe that some Android apk samples crashed during real-time execution, and this may lead to the reduction of the datasets. It will be worthwhile to investigate some strategies to avoid apk crashing during analysis. Last, we only claim our feature transformation techniques are robust to three certain types of adversarial attacks while identifying a larger space of attacks and showing our FARM work could be modified to withstand those attacks is an important future research topic.
Chapter 2

Malware Categories

Android malware is actually no different from the different forms of malware you might be acquainted with on desktop or laptop computers. It’s simply aimed at Android users. Mobile malware is some form of malicious program or code intended to damage a user’s device, such as trojans, adware, ransomware, spyware, rooting malware, or phishing applications. Android malware will come from numerous different sites. Third-party software shops, where people go to download new games, for example, hide ransomware within various forms of applications. An Android user, unlike an iOS user, can often do what’s known as sideloading an app. This does enable the system owner to modify security permission, typically called 'unknown sources'. Users will then downloading content directly from the internet to their smartphone, or from their computer, bypassing the Google Play Store entirely. In the meantime, there are several typical places Android malware will propagate like harmful downloads in emails, accessing suspicious websites, or opening links from unknown senders. When malware invades your Android operating system, it may do all kinds of things from the mischievous to the outright fraudulent, and typically can harm the user via one or more of the following behaviors:

• Show the user unwanted advertisements constantly.
Malware Categories

- Endanger the user’s system integrity.
- Spreads spam or instructions to impact other devices or networks from the infected device.
- Access or steal personal information without the user’s permission.
- Get the administrative (root access) of the device of a user.
- Allow an intruder to connect, use, or otherwise run an infected device remotely controlled operations.
- Endanger the user’s system integrity.

This chapter describes the main malware categories defined by Google Play Protect impacting users worldwide today. The effect of these app behaviors ranges widely, from fraudulently charging the user for services they did not approve or request, to the exposure of user credentials and other personally identifiable information to malicious actors.

Section 2.1

Distributed Denial of service

A denial-of-service attack is a cyber-attack in which the attacker tries to render the computer or network capability inaccessible to its intended users for a limited duration of time or forever. In a Distributed Denial of Service attack, several origins of the incoming attack assaulting the target originate. This makes it very challenging to avoid an assault merely by blocking a specific source, and a DDoS attack is like a community of people that surround the entrance door to block any people from getting inside. DDoS attackers also target web servers such as banks or credit card payment gateways, which are hosted on high-profile websites.
Malware Categories

An application that commits DoS assaults against another operating system, remote servers, or personal device formed part of a DDoS. This may happen, for instance, when applications submit a high number of HTTP requests to overburden remote servers.

Section 2.2

Backdoor

An Android backdoor malware can negate normal authentication procedures to enter a device’s core system. Anyone that has remote access through the backdoor malware can remotely issue system commands and upgrade the malware itself through the remote control. A backdoor is accomplished by leveraging vulnerabilities in a web application. When malware is mounted, it can be challenging to identify because of the strong obfuscation, and the potential malicious activities may bring the app into other types of malware categories such as spyware, phishing, and adware.

Usually, backdoors are used by hackers for a variety of illegal reasons as below:

- Intellectual Property abuse.
- Website defacement.
- Server penetration.
- Launch of distributed denial of service (DDoS) attacks.
- Disturbing website visitors (watering hole attacks).
- Advanced persistent threats or APT attacks.
Spyware is software that illegally gathers information and sends it to an attacker’s Command and Control server. Unlike ransomware (which explicitly locks a user’s screen and prevents access to files through encryption), SMS fraud (where users may notice charges), or banking trojans (where users may notice money disappearing from their bank account), spyware is stealthy and can siphon data from calendars, emails, SMSes, contact lists, social media accounts, and more, without the user becoming aware of it. Spyware also constitutes an initial “reconnaissance” phase for more complex attacks. For instance, spyware is often associated with more dangerous threats such as cyber espionage targeting top executives, whaling attacks (in which large amounts of money are siphoned off from companies), and private data theft for blackmail and extortion purposes (e.g., compromising photos).

Commercial spyware always sends the device’s personal details without giving appropriate warning or consent. Commercial spyware applications transfer data to a third party rather than the app’s creator or supplier (for instance, to the user who set up the app on the target device). Parents may use legal versions of these apps to monitor their children, but the same apps may often be used to track another user (for example, a spouse) without their consent or approval. Spyware may carry out a range of harmful activities. Spyware, for example, can use browser cookies to monitor a user’s online habits, such as queries, history, and downloads, for marketing purposes. Spyware may also take the shape of system monitors, and can log nearly everything you do on your machine. Both keystrokes, emails, chat-room conversations, websites accessed, and programs run may be registered by device monitors. Freeware is commonly used to mask system monitors.
Section 2.4

Hostile downloader

A hostile downloader is a type of malware that may not be potentially dangerous itself, but may download other potentially harmful software after its installation. If there is legitimate cause to suspect that an app was developed to distribute malware and the app has either downloaded malware or includes code that can download and install malware, the app could be identified as a hostile downloader. Major browsers and file-sharing applications aren’t deemed aggressive downloaders as long as they don’t trigger hostile downloads without user permission and don’t push downloads without user engagement (e.g., users looking for some specific rooting application for their device). Via the hostile downloader, certain trojans may be downloaded into the user’s device without the user’s awareness or permission, and when malicious programs are downloaded and executed, they can destroy, interrupt, steal the sensitive information, or break the integrity of the smartphone.

Section 2.5

Privilege escalation

Privilege escalation is a common way for attackers to breach previously expected defenses. Attackers identify weak points in an operating system’s security check procedures and manipulate them to obtain access to a device. In most instances, first of all, the initial point of entry is not enough to obtain access or core data. They can then try to elevate privilege in order to obtain access to additional system’s administrative or do additional harm. Often, attackers who seek privilege escalation find that normal security measures are insufficient, or that workers have been granted rights that they do not require. In other instances, attackers use security bugs or try to manipulate operating system weaknesses to obtain access to the system.
MALWARE CATEGORIES

An Android privilege escalation app that results in the compromise of the system’s resources by acquiring elevated privileges or removing access to key security-related functions. Often privilege escalation may involve malware that exploits passwords from other applications and stops the user from installing an app.

Section 2.6

Ransomware

Ransomware is a type of program that attempts to block access to the user’s files if a ransom is not paid. Some basic ransomware can lock the device so that it is difficult for even a competent individual to remove it. More sophisticated malware uses artificial intelligence to control the device. Ransomware can encrypt files, making them unavailable, and threatens to erase all of them until a ransom is paid. In a properly executed cryptoviral ransomware attack, retrieving the data without the decryption key is an intractable challenge – and difficult to track when cryptocurrencies such as Bitcoin are used for the ransoms, rendering tracking, and punishing the offenders difficult. Ransomware attacks are normally carried out by viruses posing as a legitimate file, which trick users into uploading or opening the file when they receive it in an email.

An Android Ransomware app takes partial or extensive control of a computer and demands money or access to such information. Some ransomware programs encrypt data and demand payment to recover the data, and/or leverage the system administrator functionality to guarantee that the ransomware cannot be deleted by a normal consumer.
Exploitable malware that takes over Android devices to gain root access and completely control the phone. Once the mobile platform has been compromised, the application can install other malicious programs and steal confidential information. Rooting malware also tends to make its way onto devices through e-mail spam and phishing sites. When rootkits are not destructive, they act as a means to deliver other programs to the infected system.

Android rooting malware can enable the user to escalate privileges against the device’s operating system. There is a difference between unauthorized and unauthorized root applications. With non-malicious rooting apps, users are notified in advance that they are going to root a device and some apps do not have malicious features. As well, malicious rooting applications are known not to inform the user of the rooting.

A Trojan is a form of malicious code or program that pretends to be legitimate but can actually do harm through control the user’s device. A Trojan can infiltrate and damage, disrupts, steal, or in general, imposes some harm on a user’s data or on the network. Trojans function like a genuine program or file to trick the user. This type of malware tries to trick the user into running it on their device. Once installed, the Trojan can accomplish what it was manufactured to do. Trojans can infect both desktop computers and laptop computers. They can also impact mobile devices and devices including smartphones and tablets.

Android Trojans are usually disguised as what appears to be a legitimate application. In fact, it is a fake application with added malicious content. Cybercriminals will also
put pirated apps on unauthorized and piracy app markets. Furthermore, these apps can also steal information, and generate income by sending premium text messages. Trojan malware has been particular targeted at Android devices, and they allow cybercriminals to infect users’ devices and redirect users’ traffic through their home Wi-Fi connections, allowing them to commit various crimes.

Section 2.9

SMS fraud

SMS fraud involves the distribution of malware by cybercriminals designed to target a victim’s mobile device. After SMS fraud malware is installed on the user’s smartphone, it makes unwanted phone calls and text messages without the user’s permission or consent. These illegal calls or SMS messages are then routed to premium-charge number services operated by cybercriminals, which creates substantial revenue sources for cybercriminal networks.

Section 2.10

Spam

Spam is defined as messages sent to users of the internet with the motive of advertising, phishing, or spreading a virus. Email spam is usually sent in bulk to users, the common usage of the term. Spams come in various forms including unsolicited messages in instant messaging, blog comments, social media posts, and mailed advertisements. Those who regularly participate in mass email campaigns may upload an excel file to the mailing system and send thousands of emails in the blink of an eye.

An Android spam application can send unwanted messages to the user’s contacts or utilize the device as an email spam transmitter. Although it may not be specific Android threats and these apps may not harm Android users or devices, they potentially
Malware Categories

contain harmful components for other platforms like iOS or Windows.

Section 2.11

Toll fraud

Toll fraud is a scheme where fraudsters conduct high-volume transactions involving international calls through costly routes. Fraudsters dial a premium number and take a share of the revenue created from calls to these numbers. Toll fraud involves three ingredients: a lengthy and opaque value chain, misuse of a service that is given free of charge, and kickbacks from the party collecting payments to the party sustaining the service. In toll fraud, fraudsters share revenue with foreign premium number suppliers and businesses that do business by buying and reselling number ranges from carrier aggregators or directly from country regulators. Then these numbers are sold to consumers who can use them as content utilities, such as adult chat lines, tech assistance, and online voting.

A mobile toll fraud application may obtain subscriptions to utilities and products using a user’s mobile phone bill. Toll fraud involves many forms of billing, and Wireless access point (WAP), Mobile Airtime Transfer, and Direct Carrier Billing are technology widely exploited in these forms of fraud. In particular, WAP fraud is one of the most common forms of toll fraud which may involve tricking users into pressing a button on a quietly loaded WebView, resulting in the user accidentally activating a recurring subscription. A verification text or email is always blocked in order to avoid alerting the victim to the scam.
Chapter 3

Static and Dynamic Features of Android Applications

In this chapter, we include a quick summary of some static and dynamic features that are widely used for Android malware detection in machine learning models.

Static analysis helps one to fully analyze all the content in the kit of the Android framework (including source code and other resources such as images, strings, XML files, and so on). Static analysis, though, may provide false results as many code execution paths may never be executed. Dynamic analysis, on the other hand, fills the void by running the application in a controlled and isolated setting (e.g., an emulator or a sandbox) that helps one to see what the application is doing in operation and gather behavioral data during real-time execution, such as the behavior of network traffic, files read and written or binary files dropped. The malware developer could have used anti-analysis strategies to identify the possibility that the application is operating within an emulator or being debugged, and then adjust the behavior of the application to mask any harmful behavior.

There are benefits and drawbacks of all approaches and they are complimentary. Static and dynamic features created by static and dynamic analysis, respectively, are typically combined while using machine learning methods for malware detection in
order to optimize detection accuracy—not only on the Android platform but also on other platforms.

In Section 3.1, we first address the static features of Android applications. We begin by presenting the file structure of Android applications that are disassembled. In the disassembled code file (Smali code), we then define a collection of static features that can be collected from the content. The Section 3.2 briefly describes the dynamic analysis mechanism and then gives information on dynamic features that are generally correlated with Android applications.

### Section 3.1

**Static Features**

Until application is assembled into an executable format, it is a set of different files and directories. When investigating malware, we always find ourselves inspecting executable code that has been assembled into the .dex file format. These executables contain bytecode that is translated and executed within the ART or Dalvik runtimes. The bytecode cannot be interpreted by humans, so they need to be converted into a more understandable script, such as Smali code (disassembling) or Java code (decompilation). Bytecode can then be disassembled relying on software such as APKtool and JADX to get the source code for analysis. The analysis of Android applications is achieved without needing to execute the applications because of the static contents of application files. This contributes to static analysis’s attractiveness as an approach.

Inside the source files of an Android application, one can find the following constituent items:

- **Manifest** Each Android application includes a `AndroidManifest.xml` file in its root directory. The manifest file determines the structure and metadata of the Android application. The manifest provides essential details of the application.
For example, the package name, which is conventionally a unique identification of the application, file size, and version are included in the manifest. The manifest also includes XML nodes that describe the basic behavior of applications. In addition, permissions requested by the application are also listed in the manifest file. Figure 3.1 shows the snapshot of the manifest file of a variant *Zsone.245* of the Zsone trojan which has the potential to charge for SMS messages to premium-rate numbers.

- **Java** Android applications are typically built with Java programming language that is stored in the JAVA folder.

- **Res** The res folder provides a range of tools outside the code itself such as XML layouts and images used in the application.

- **Build.gradle** This file includes build-related configurations of the application.

A collection of static features can be derived from an Android application such as by scripts that automatically extract certain features. We now introduce several generally used static features as below:

---

1 More details on this particular trojan are available on VirusTotal [https://www.virustotal.com/gui/file/3274445785390407159794d4085b12fb400d1f4d56a0707409c612dd7a21411/community](https://www.virustotal.com/gui/file/3274445785390407159794d4085b12fb400d1f4d56a0707409c612dd7a21411/community)
• **Permission Features** Permissions must be required by Android applications to access confidential user data (e.g., messages) or some interface functions (e.g., to send SMS messages). It is necessary to specify these permissions in the manifest file. We may develop a range of features relevant to the permissions that the application requests. For instance, one can create a "permission:NAME" binary function, with "NAME" being substituted by the name of each permission. This form of feature indicates whether an application requires the corresponding permission. Permissions can be divided into three levels of security according to the guidelines on the official Android developer website regular-dangerous, normal permissions, signature permissions, and dangerous permissions. Dangerous permissions are those that either include users’ private data or could potentially impact certain data of private users. For instance, the ‘READ CONTACTS’ permission, which grants the application access to user’s contacts stored on the device, is categorized as dangerous permission. Static features that correspond to the total number of the normal/signature/dangerous permissions required by the application can also be specified in the feature vector. For example, a version of the Arspam Android Trojan malware that promotes a form of political activism named Alsalah.cf.rm.renaming requests a very large number of permissions, part of which are shown in Figure 3.2. Many of these permissions are used to propagate the malware to people on the compromised user’s contact list.

• **Activity** Activities incorporate an Android application’s interactive interface and are defined in the manifest file as well. In order to show whether they are

---

1. https://developer.android.com/guide/topics/permissions/overview#regular-dangerous
2. More detail on Arspam can be found at: https://androidcommunity.com/android-arspam-is-the-latest-malware-threat-says-symantec-20111230/
3. More details on VirusTotal through https://www.virustotal.com/gui/file/0a0ff315ef11478fc3f8e08af7312f05430c6a977f246d88224b812c948c7056/detection
used or not, a collection of binary features can be generated for each activity. Also, a potential feature is the total number of activities. As we can see from the screenshot, for example, Figure 3.1 \textit{Zsone}_{245} contains activities such as hiding the keyboard.

- **Services** In order to promote the interaction between the application and the system and also with other applications, services are used to execute long-running background operations. We can specify a binary feature (used or not) for each service, in the same way as with activities. In addition, it is also possible to define basic count and advanced statistical features based on that. Figure 3.3 displays a portion of the trojan malware sample manifest file \textit{test.txt}. In its manifest, this malware application has 3 separate services, 2 of which are seen in the figure, i.e., line 52 and line 79 of the file.

- **Content Provider** To encapsulate data and supply it to other applications, a content provider is defined and used. It is also possible to specify binary features and statistical features for providers in a way that is close to that mentioned...
above. For example, to decide what content providers exist and how many are utilized by the application.

- **Broadcast Receiver** This part of an application requires intents from the device or other applications to be obtained. It is also possible to specify binary features and statistical features for providers in a way that is close to that mentioned above. Some of the receiver examples are seen in Figure 3.3. For example, the receiver in Lines 66 — 70 shows that after booting, the application will receive the intent sent by the system.

- **Intent Filters** In order to determine what kind of operations they should react to, activities and services involve intents. Intents determine what broadcast they should accommodate. Similarly, it is possible to derive binary features and statistical features for intent. Some examples of filter intents are seen in Figures 3.1 and 3.3. Lines 68, for instance, suggests that the application requests a
"BOOT_COMPLETE" intent, i.e. that the booting mechanism is complete.

- **API calls** These consist of a collection of application programming interface (API) packages for developers to construct applications is provided by the official Android framework. For certain API packages and their classes specified in the source code, binary or integer features can be extracted. In the following chapter, we include a thorough introduction to API features as they are used to create more advanced features. For example, the malware sample Zsone_245 called *android.hardware.fingerprint* API as shown in Figure 3.4.

- **Hardware** One or more hardware components of Android devices might be used for Android applications. Sensors such as the phone’s camera, GPS, and speaker provide such hardware components. It is likely that this detail is declared in the manifest file, although that is not necessarily the case. Some applications can require these permissions in the app’s source code instead of in the manifest file. The associated feature set is called using the format "hardware:NAME" with "NAME" replaced with the name of the hardware component. Hardware feature-associated permissions can be coded as binary features with a 1 denoting that the application uses the corresponding hardware component and a 0 implying that it does not. Similarly, to describe a static feature, the number of such hardware permissions required can also be used. Zsone_245 requires access to a number of hardware components (including *fingerprint* as shown in Figure 3.4) of the Android device without the permissions declared in the manifest file. Instead, from the source code, we find those specifications.

- **Network elements** Numerous network components, e.g. IP addresses, URLs, and hostnames, can be used in an android application. To produce binary features and statistical features, these elements are obtained from the source code (e.g., the number of hostnames that are listed in the file). Figure 3.5
suggests that the Zsone 245 malware sample contains a malicious URL that is used to view photos such that advertising revenue can be obtained from ad clicks developed by the malware developers.

**Section 3.2 Dynamic Features**

Android applications reveal their run-time behavior in dynamic analysis. A controlled and isolated environment called a sandbox used for running potential harmful applications.

In order to dynamically analyze an Android application, we mount the application in a sandbox environment first. To replicate the use of the application on real devices by real users, a sequence of commands is then sent to the sandbox environment. In order to recognize the operations that are thereby activated in the application, for example, the system can be rebooted.

---

6There is a range of sandboxes for Android applications to deter harmful applications causing possible damage to other applications or the device during the execution of the potential harmful app. Two common sandboxes, Droidbox [https://github.com/pjlantz/droidbox](https://github.com/pjlantz/droidbox) and Cuckoo [https://cuckoosandbox.org/](https://cuckoosandbox.org/)
Usually, we run the application for a fixed time window inside the sandbox (e.g. 2 - 10 minutes) and save the log/trace of the activity of the application, providing a list of all device processes, connections/calls to other systems, as well as the expected network interactions. In order to produce dynamic features, we then extract information from the trace. There are many businesses that offer facilities in different sandboxes to test Android applications and produce reports from their run-time logs. These reports will then be used for the extraction of dynamic features.

A variety of dynamic features from similar works are briefly presented and discussed below [63, 64, 65, 66, 67, 68, 61, 62].

- **Service** We can create dynamic features to record each started service with NAME as ”service:NAME”, and the cumulative number of services started can also be set as a feature. An n-gram of services is a series of services. One class
of dynamic features shares one feature with each $n$-gram of services. Given a $n$ gram $s_1, \ldots, s_n$ of services, we can record whether this sequence is ever invoked in the runtime execution of the code. Figure 3.6 shows that a malware sample named Plankton\(^7\) is executed in Droidbox. It begins the notification service on two different occasions within 0.01 second. Plankton is secret spyware that extracts information from the user’s internet browser, including bookmarks, search history, and passwords.

- **Dexclass** *DexClassLoader* is a public class used to load classes from every form of .jar, .zip, and .application bundles. Malicious applications can run code on a user’s device without being installed as part of the application. Whenever the class loader *DexClassLoader* is called to load a class, a dynamic feature ”dexclass:NAME” can be generated. Similarly, statistical features and $n$-gram based features can be created. Figure 4 shows that Plankton use *DexClassLoader* to load classes from *input.jar*.

- **Permissions** Although Android applications must specifically declare the permissions they request in their manifest file, Android applications can still attempt to bypass this requirement in different ways\[^69\]. The easiest approach is to use

\[^7\]More details can be found on VirusTotal via https://www.virustotal.com/gui/file/00ceaa5f8f9be7a9ce5ffe96b56fb2e7e73ad87c2f023db9fa399c40ac59b62/detection
the ”convert channel” to exchange information between different applications. The circumvented permissions are monitored and features ”permission:NAME” are created. As in the previous situations, statistical features and n—gram based features can be created. Figure 3.8 presents part of the permissions requested by the Plankton malware while running in Droidbox.

- **Dataleaks** The operations conducted by an application could contribute to the user’s personal data being exposed. Such actions may be documented to create features such as ”dataleaks:CONTENT”. Figure 3.9 indicates that a device’s IMEI details may be leaked.

- **Crypto** Also, we can connect a feature that monitors the success of private encryption keys in Android applications. If an application encrypts files and saves them in a sandbox, the device monitors and documents them. These features are used to create dynamic, content-specific cryptographic features in
the format "crypto:CONTENT".

- **Opennet/Closenet** The socket related features maintain records of where and how sockets are opened and closed. We may call the destination host either "opennet:DEST" or "closenet:PORT".

- **Recvnet** This set of features help maintain track of the data collected from the network("recvnet:CONTENT") and the origins of the data("recvnet:FROM"). One of the recvnet activities of the *Plankton* malware is shown in Figure 3.10.

- **Sendnet** This set of features maintain the track of how the application sends ("sendnet:CONTENT") and received information. The destination ("send-
Static and Dynamic Features of Android Applications

Figure 3.11: Sendnet operation by Plankton in Droidbox

```
759  "detail": {
760       "data": "504f5354202f50726f746f636f6c47572f70726f746f636f6c2
761           f636f6d6d616e6473204544502f312e3100a646576696362d69643a2
762           066366458426464554e376266423576473768635a326c632532466839552
763           5334400a70726f746f636f6c2d76",
764       "desthost": "192.168.56.101",
765       "destport": "80",
766       "fd": "42",
767       "operation": "send"
768   },
769       "process": "/init-zygote->com.androidpeople.tab.freeplayer",
770       "time": 13.391,
771       "type": "SendNet"
772   },
```

Figure 3.12: Receive actions by Plankton in Droidbox

```
49   "recvsaction": {
50       ".BootReceiver": "android.intent.action.BOOT_COMPLETED",
51       "com.moolah.BootReceiver": "android.intent.action.BOOT_COMPLETED"
52   },
```

net:DEST”) is also recorded which can later be extracted as dynamic features. One of the recvnet activities of the Plankton malware is shown in Figure 3.11

- **Sendsms** We will document the name of the receiver of messages and their content in situations where applications send messages out when they are operating. Records of “sendsms:NUM” and “sendsms:MESSAGE” are generated as dynamic features to collect related information.

- **Phonecall** Malicious Android applications occasionally make phone calls (e.g. to premium-rate numbers). When specifying features such as “phonecall:NUM”, we save the telephone number of the called number.

- **Recvsaction** It is also possible to catch the list of intents the application refers to during execution as dynamic features. Two intents associated with the Plankton malware are captured and shown in Figure 3.12.

- **Library** This documents the features associated with the application’s library directories, referred to as “library:NAME” features.
Read/Write The endpoint and the content will be captured while the application reads or writes individual files. Then we create the "read:DEST", "write:DEST", "read:CONTENT" and "write:CONTENT" features. Figure 3.13 presents one file read operation of Plankton in Droidbox after the application has run for 3.91 seconds.
Chapter 4

Advanced Features of Android Applications

In this thesis, the advanced machine learning algorithms we use to identify between benign and malicious Android applications all associate a "feature vector" with each Android application. We implement four new types of features (Triadic Suspicion Graph based Features in Section 4.2, Landmark-based Features in Section 4.3, Feature clustering-based Features in Section 4.4, Correlation Graph based Feature Transformation in Section 4.5) in this chapter, in addition to the typical types of static and dynamic features used in past work and defined in Chapter 3.

Section 4.1

API Features

An application programming interface (API) for an application is a collection of subroutine definitions, communication protocols, and software construction methods. There is a collection of API packages for Android apps that developers can use to access a variety of useful functionalities. The API Package, for instance, android.accessibilityservice can build accessibility services that provide up-
Advanced Features of Android Applications

dates to the user. A variety of classes are included in each API package and each class has its own methods that execute various functions.

API packages are not totally independent, which ensures that by calling classes and methods from other API packages, they can communicate with each other. For instance, the android.provider API package invokes classes from the android.view package.

The Android API (level 23) includes 171 various API packages. The 171—dimensional feature vector $\tilde{f}$ is designed for each Android app to capture the frequency at which that application calls methods from each API package. Each $f_i \in \tilde{f}$ unit represents the number of method calls made to methods in the $i$th API package in the source code of that program. Through evaluating the app’s source code, we obtain the feature values. For example, if the $i$th API package contains 40 methods (belonging to separate classes) and each of them is called by an app twice, then 80 will be the corresponding feature value of $f_i$.

An application’s API feature values will differ greatly. Consider a goodware sample named OptiLife 2.8.3. For instance, when considering a goodware sample (i.e. a regular Android program without malicious behavior), the highest value function has a 389,204 value, while the smallest feature has a 0 value among all the 171 API features associated with OptiLife 2.8.3. On this one application only, the standard deviation of feature values is as high as 32,333.47.

Conversely, for other applications, the distribution of API feature values can be very different. For example, another sample of goodware called Hancom Office 2014 has the largest API value of 6 and the smallest of 0, with a standard deviation of only 0.61.

1SHA256: 0e42318be9faea983330723162b4c7c3d5ccc97adf83b0eb0e4127bb89929
2SHA256: 75d1ad83a77d1d66d17586d7f83cb4e56e136b5cc636002f5555f434819b162
Section 4.2

Triadic Suspicion Graph based Features

In this section, we present a novel set of features that I have invented based on a special class of graphs first defined in [71].

4.2.1. Triadic Suspicion Graph

The notion of a Triadic Suspicion Graph (TSG) is based on API features and will be used to describe a collection of novel features shortly.

The fundamental concept behind a TSG is to consider the differences between goodware’s use of API package calling behavior and various types of malware’s usage of API package.

Suppose \( A \) is the set of all API package calls defined within the Android API, \( \mathcal{G} \) is a set of Android goodware, and \( \mathcal{B} \) is some type of malware we interested in (in subsequent chapters, these additional types of malware will include spyware, ransomware, banking trojans, and more). Note that both \( \mathcal{B}, \mathcal{G} \) are samples that are randomly drawn from the set of malware of interest and the set of goodware. We use \( \text{TSG}^{\mathcal{G}, \mathcal{B}} \) to denote the triadic suspicion graph associated with \( \mathcal{G} \) and \( \mathcal{B} \). The triadic suspicious graph contains three kinds of vertices: API package calls of set \( A \), goodware drawn from the set \( \mathcal{G} \) and malware from the set \( \mathcal{B} \). The edges of \( \text{TSG}^{\mathcal{G}, \mathcal{B}} \) are defined as follows:

(a) For each goodware \( g \) in \( \mathcal{G} \) and each API package call \( a \) in \( A \), if \( g \) calls a method from \( a \) at least for once, then there is an edge from \( g \) to \( a \).

(b) For each pair of API package calls \( a_1 \) and \( a_2 \) in \( A \), if \( a_1 \) calls any method from \( a_2 \), there is an edge from \( a_1 \) to \( a_2 \).

(c) For each malware \( b \) in set \( \mathcal{B} \) and each API package call \( a \) in \( A \), if \( b \) calls any method from \( a \) at least once, we add an edge from \( b \) to \( a \).
We do not require that $G$ and $B$ be fixed. A system security analyst might use $G_1$ and $B_1$ in one week, switch to $G_2$ and $B_2$ the next week, and keep doing so regularly in order to present a “moving target” defense. Varying the sets $G$ and $B$ changes the attack surface adjusts and thereby makes it difficult for the adversary to presume or speculate too much about the defense’s exact nature. Furthermore, we suggest keeping the size of $G$ and $B$ relatively small. For example, if there are $1M$ goodware samples and $10K$ malware samples in total, we might select only say $1K$ samples for each of $G$ and $B$ one week, $1.5k$ the next week, $2k$ the third week, and so forth. By frequently modifying the sizes and samples in $G$ and $B$, the defender is keeping the attacker continuously guessing about the nature of the defenses being used.

Once the vertices and edges in a TSG are determined, we additionally allow the edges to be weighted by a weight function $w$. For any edge from an application $v \in G \cup B$ to an API package call $a \in A$, we use $f(v, a)$ to denote the frequency (number of times) that $v$ calls methods from $a$ (which is also the corresponding API feature value). Equations (4.1) to (4.5) demonstrate 5 plausible definitions of weight functions $w$. It is easy to see that Equations (4.1), (4.2) and (4.3) respectively represent linear, quadratic and cubic relationships between the API package call frequency and the edge weight; meanwhile, Equations (4.4) and (4.5) capture other possible non-linear relationships. The intuition behind these different definitions is that most machine learning algorithms are very sensitive to the input features and cannot always correctly infer the best non-linear relationships using the modeling framework alone. We set the weight of edges between pairs of API package calls to the same default value (e.g., 1). The reason is that we are more interested in whether a specific edge exists, rather than the frequency with which one API package calls the another within the Android.
Advanced Features of Android Applications

API as this is controlled by Google’s Android team and not by malicious hackers.

\[ w_1(v, a) = f(v, a) \]  \hspace{1cm} (4.1)

\[ w_2(v, a) = f(v, a)^2 \]  \hspace{1cm} (4.2)

\[ w_3(v, a) = f(v, a)^3 \]  \hspace{1cm} (4.3)

\[ w_4(v, a) = \sqrt{f(v, a)} \]  \hspace{1cm} (4.4)

\[ w_5(v, a) = \ln \left( f(v, a) + 1 \right) \]  \hspace{1cm} (4.5)

We use \( \text{TSG}^{G,B,w} \) to denote the triadic suspicion graph that is generated using \( B, G \) and the weight function \( w \).

Figure 4.1 demonstrates part of a TSG built with some goodware samples, a set of banking trojans and the weight function \( w_1 \) in Equation (4.1). 3 goodware samples (Perfect Girls\(^3\), Azerbaijan Radio World\(^4\), and Iberia Persseo\(^5\)), 3 banking trojan samples (Regon\(^6\), Marcher\(^7\), and Fakebank\(^8\)) and 4 API packages (android.view, java.net, android.app.admin and java.util) are shown in the figure (in this scenario, the full TSG will contain all the package calls present in the Android API and potentially a lot more samples of goodware and banking trojan samples). The

![Illustration of TSG](image)

\(^3\)SHA256: 00061593f5b0299bd46c5c1db502319c8274ba8c206d7abc0b59d7b23ab7e
\(^4\)SHA256: 0009666ca8eaac88f23ba93974ec5964243ae839382ae9683e5d1062521f240
\(^5\)SHA256: 0019607c1427d57c1ec99edcfde546c725e8e4297a10ce8b4956be72ca061fc4
\(^6\)SHA256: 0007c4e543e610f0b0e933545bd927d9c8078380aa3518ce807bb9e0989903
\(^7\)SHA256: 0010cc30886b77d922de13230a9df16de11463b5be79ce4a0d36d26d6555bd0348
\(^8\)SHA256: 002ef1caac9d5e567660e6f6001d39863c526bfa292a764ff241c08503956fb2
edges from Android applications to API packages are represented with gray arrows, where the numbers denote the number of times that they call the corresponding API package’s methods (i.e., this number is the same as its corresponding API feature value defined previously in Section 4.1). Green arrows in the figure show the calling relationship amongst pairs of API packages. As we observe, none of the 3 goodware samples call the API package android.app.admin, while 2 out of the 3 banking trojans call it a few times. The intuition behind our work is that this pattern might be relevant in distinguishing goodware from banking trojans.

We now define a set of novel features based on TSGs.

4.2.2. Suspicion Score

With the TSG defined, we now introduce the concept of suspicion score (SUS) of API packages in $\mathcal{A}$. The idea is that an API package that is frequently invoked by malware but not by goodware is more suspicious than one that is frequently invoked by goodware, but not by malware. We do not claim that suspicion scores alone are enough to predict whether an Android application is malicious or not, just that it provides a set of features of interest.

First, we introduce an indicator function $I(v_1, v_2)$ to denote the existence of an edge from vertex $v_1$ to $v_2$, where $v_1, v_2 \in \mathcal{A} \cup \mathcal{G} \cup \mathcal{B}$. In other words, if it is the case that $f(v_1, v_2) > 0$, then $I(v_1, v_2) = 1$, otherwise it is 0. Actually, $I(v, a)$ for edges from benign/malicious applications to API packages can be treated as another kind of weight function $w_0(v, p) = 1$ if $f(v, a) > 0$ else 0. In the following we use $n$ to denote $|\mathcal{B}|$ and $m$ to denote $|\mathcal{G}|$. We list 12 definitions of SUS (Equations (4.6) to (4.17)). The reason for multiple definitions is that each of these definitions represents one way of defining suspicion levels. By defining multiple candidates, we are less worried about over-fitting a predefined model. Moreover, using these 12 suspicion score definitions and other features as input, machine learning techniques can tell us which suspicion
scores are best able to differentiate benign applications from malicious ones.

\[
sus(a_j) = \frac{\sum_{i=1}^{n} f(b_i, a_j)}{\sum_{i=1}^{n} f(b_i, a_j) + \sum_{i=1}^{m} f(g_i, a_j)}
\]  \hspace{1cm} (4.6)

\[
sus(a_j) = \frac{\sum_{i=1}^{n} f(b_i, a_j)^2}{\sum_{i=1}^{n} f(b_i, a_j)^2 + \sum_{i=1}^{m} f(g_i, a_j)^2}
\]  \hspace{1cm} (4.7)

\[
sus(a_j) = \frac{\sum_{i=1}^{n} f(b_i, a_j)^3}{\sum_{i=1}^{n} f(b_i, a_j)^3 + \sum_{i=1}^{m} f(g_i, a_j)^3}
\]  \hspace{1cm} (4.8)

\[
sus(a_j) = \frac{\sum_{i=1}^{n} \sqrt{f(b_i, a_j)}}{\sum_{i=1}^{n} \sqrt{f(b_i, a_j)} + \sum_{i=1}^{m} \sqrt{f(g_i, a_j)}}
\]  \hspace{1cm} (4.9)

\[
sus(a_j) = \frac{\sum_{i=1}^{n} \ln(f(b_i, a_j)+1)}{\sum_{i=1}^{n} \ln(f(b_i, a_j)+1) + \sum_{i=1}^{m} \ln(f(g_i, a_j)+1)}
\]  \hspace{1cm} (4.10)

\[
sus(a_j) = \frac{\sum_{i=1}^{n} f(b_i, a_j)}{\sum_{i=1}^{n} f(b_i, a_j) + \sum_{i=1}^{m} f(g_i, a_j)}
\]  \hspace{1cm} (4.11)

\[
sus(a_j) = \frac{\sum_{i=1}^{n} f(b_i, a_j)^2}{\sum_{i=1}^{n} f(b_i, a_j)^2 + \sum_{i=1}^{m} f(g_i, a_j)^2}
\]  \hspace{1cm} (4.12)

\[
sus(a_j) = \frac{\sum_{i=1}^{n} f(b_i, a_j)^3}{\sum_{i=1}^{n} f(b_i, a_j)^3 + \sum_{i=1}^{m} f(g_i, a_j)^3}
\]  \hspace{1cm} (4.13)

\[
sus(a_j) = \frac{\sum_{i=1}^{n} \sqrt{f(b_i, a_j)^2}}{\sum_{i=1}^{n} \sqrt{f(b_i, a_j)^2} + \sum_{i=1}^{m} \sqrt{f(g_i, a_j)^2}}
\]  \hspace{1cm} (4.14)

\[
sus(a_j) = \frac{\sum_{i=1}^{n} \ln(f(b_i, a_j)+1)^3}{\sum_{i=1}^{n} \ln(f(b_i, a_j)+1)^3 + \sum_{i=1}^{m} \ln(f(g_i, a_j)+1)^3}
\]  \hspace{1cm} (4.15)
\[
sus(a_j) = \frac{\sum_{i=1}^{n} \sqrt{f(b_i, a_j)}}{\sum_{a_j} \sum_{i=1}^{n} \sqrt{f(b_i, a_j)} + \sum_{a_j} \sum_{i=1}^{n} \sqrt{f(g_i, a_j)}} \quad (4.16)
\]

\[
sus(a_j) = \frac{\sum_{i=1}^{n} \ln(f(b_i, a_j)+1)}{\sum_{a_j} \sum_{i=1}^{n} \ln(f(b_i, a_j)+1)} + \frac{\sum_{i=1}^{m} \ln(f(g_i, a_j)+1)}{\sum_{a_j} \sum_{i=1}^{m} \ln(f(g_i, a_j)+1)} \quad (4.17)
\]

It is easy to see that the different definitions of SUS are closely related to the weight functions \((w_0 \text{ to } w_5)\). We discuss a couple of these suspicion scoring functions below. For example, the first SUS definition in Equation (4.6) basically means that the suspicious score of the API package \(a_j\) is decided by the ratio of the percentage of malware applications in \(B\) that call it to the sum of the percentage of malware applications in \(B\) that call it and the percentage of benign applications in \(G\) that call it. Meanwhile, the SUS definition in Equation (4.12) takes Ratio \(B\) — the ratio of the number of malicious applications in \(B\) that invoke \(a_j\) to the sum of the number of malicious applications that invokes each API packages divided by the sum of Ratio \(B\) and Ratio \(G\) — the ratio of the number of benign applications in \(G\) that invoke \(a_j\) to the sum of the number of benign applications that invokes each API packages, as the suspicious score of the API package \(a_j\). Equations (4.12) to (4.17) are respectively similar to equations (4.6) to (4.11) except that the suspicion score of one API package is evaluated w.r.t. all API packages rather than itself alone.

### 4.2.3. Suspicion Rank

Furthermore, inspired by the definition of PageRank\(^7\), we define a concept — suspicion rank \(SR(a)\) for an Android API package \(a\) w.r.t. a suspicion scoring function

\(^{7}\)PageRank is a mechanism to capture the importance of webpages using the formula \(PR(v) = \frac{1-d}{N} + d \times \sum_{(u,v) \in E} \frac{PR(u)}{out(u)}\), where \(E\) is the set of edges in the web, \(N\) is the total number of nodes in the web, \(d \in [0, 1]\) is called the “damping factor” and is usually set to 0.85, and \(out(u)\) is the out-degree of node \(u\). The \(\frac{1-d}{N}\) expression captures the probability that a user will reach webpage \(v\) directly (e.g., by typing it in explicitly into a browser) while the \(d \times \sum_{(u,v) \in E} \frac{PR(u)}{out(u)}\) is intended to capture the probability of a user reaching page \(v\) by following links.
**Advanced Features of Android Applications**

$sus$ as Equation (4.18). The parameter $\delta \in [0, 1]$ is a “damping factor” similar to $d$ of PageRank. In practice, we set $\delta = 0.85$ as is usually done with PageRank. Besides, $out(a')$ is the out-degree of API package $a'$ in $TSG^{B,G,w}$ (i.e., the number of API packages invoked by $a'$). Note that only API packages $a'$ that invoke $a$ are taken into consideration in calculating $a$’s suspicious rank (SR).

$$SR_{sus}(a) = \frac{1-\delta}{|A|} + \delta \sum_{a'\in A, (a',a)\in E} \frac{su(a') \cdot SR_{sus}(a')}{out(a')}$$

Note that the SUS function $sus(a')$ in the above equation defining $SR$ can be any of Equations (4.6) to (4.17). Therefore, it is noteworthy that SR is obviously different from PageRank — instead of one definition, SR has a family of 12 different definitions, each corresponding to one definition of SUS.

Since SUS is a measure defined for API packages on the complete $TSG^{B,G,w}$, and the sets $B$ and $G$ are, in some sense, dynamic, SUS and SR would vary accordingly when security analysts update the members of $B$ and $G$.

The definition of SR mainly relies on a sub-graph structure of $TSG^{B,G,w}$, i.e., the sub-graph with vertices in the set $A$ of API packages and the edges among them. We name this sub-graph “The Android API package call graph (APCG for short)” and provide more discussion on it in the rest of this section.

The sub-graph APCG of the triadic suspicious graph only depends on the set $A$ of API packages and the mutual calling behavior among them. Thus the structure of the APCG is independent of the choice of applications in the sets $B$ and $G$. In other words, irrespective of how $B, G, w$ are chosen, the APCG part of $TSG^{B,G,w}$ of different $B, G, w$ would remain unchanged. There is no possibility for adversaries to manipulate the APCG. This property makes it different from existing works with function call graphs.

Function call graphs in existing works, like dependency graphs, control-flow
graphs [74, 9] and code property graphs [75] usually rely on specific application samples and depend on the sequence of operations within the applications. In other words, prior works abstract per-app function call graphs, which are then used by detection models to identify the similarity among applications. In contrast, we build the APCG based on the universal API packages without touching any specific applications, then a single large TSG graph (Figure 4.1) is developed on the basis of the APCG with additional sets \( B \) and \( G \) of applications. This TSG is further used to derive a family of SUS and SR for each API package in the APCG. We can then generate TSG features according to API packages’ SUS and SR for any Android app, irrespective of whether it is a member of \( B \cup G \) or not. One single graph for all applications as compared to one graph per app significantly improves efficiency and generality.

Recall that we have previously recommended that the sets \( B \) and \( G \) should be dynamically changed over time. Because of this, our proposed approach also possesses the advantage of establishing a “moving target” defense [76].

4.2.4. Window-based TSG Feature Creator

The preceding two sections define SUS and SR for API packages in triadic suspicious graph. Clearly, there are in total \( 12 + 12 = 24 \) kinds of suspicion-based scores associated with each API package.

In the next step, we use the suspicion-based scores derived from a TSG to generate what we call “TSG features” for Android applications. One thing to note is that TSG features are used to capture the API package call behavior of applications, and the application does not have to be in either the set \( B \) or the set \( G \) to have TSG features.

We use \( \rho \) to represent a SUS score function or a SR rank function. The API packages in \( A \) are ranked in descending order according to their scores returned by the function \( \rho \). One way to think about the ranked API packages is that the higher rank of the API package, the more suspicious it is. However, we probably have noise
(which might come from the choice of sample applications in sets $\mathcal{B}, \mathcal{G}$) in computing SUS or SR for API packages. Therefore, instead of directly using the ranked API package list, we apply a window-based segmentation on the list before deriving TSG features.

The basic idea of window-based segmentation is to use an integer $W > 1$ as the window size and segment the list into a number $\left\lceil \frac{|\mathcal{A}|}{W} \right\rceil$ of buckets starting from the beginning of the list. As shown in Figure 4.2, each bucket (except possibly the last one) has $W$ API packages with similar suspicion-based score or rank.

![Figure 4.2: Window-based API package ranking by descending SUS/SR scores.](image)

Recall that we define a $|\mathcal{A}|$-dimensional API feature vector $\tilde{f}$ in Section 4.1 for each application to capture its basic API package call behavior. Now for each Android app, we define a $\left\lceil \frac{|\mathcal{A}|}{W} \right\rceil$-dimensional TSG feature vector $\tilde{f}^{\text{tsg}}$ according to $\tilde{f}$ and the suspicion of API packages. Assume that API packages $a_1, \ldots, a_W$ are in the same bucket, the corresponding API feature values of an application are $f_1, \ldots, f_W$, the corresponding TSG feature for this bucket is then calculated via one of the following 6 methods:

(a) (Binary Value) Does this application call any API packages from this bucket? This binary feature is 1 if $\sum_{j=1}^{W} f_j > 0$, else 0.

(b) (# of API Packages) How many API packages in this bucket are called by the app? The feature value is an integer $\sum_{j=1}^{W} I(f_j)$, where function $I(f_j) = 1$ if $f_j > 0$, else 0.

(c) (Max) Among the call frequencies (the number of calling times) from the
application to all API packages in the bucket, what is the maximum value? The feature value is an integer \( \max_{j=1}^W f_j \).

(d) (Median) Among the call frequencies, what is the median? The feature value is an integer \( \text{median}_{j=1}^W f_j \).

(e) (FreqSum) How many times in total does this application call API packages in this bucket? The feature value is an integer \( \sum_{j=1}^W f_j \).

(f) (WtSum) Based on FreqSum, what would be the value if we take the suspicious-score given by function \( \rho \) as the corresponding weight? This feature is a real value \( \sum_{j=1}^W \rho_j f_j \), where \( \rho_j \) stands for the suspicious-score of API package \( a_j \).

To illustrate how this works, consider the small dataset with 3 banking trojans and 3 goodware samples in Figure 4.1. Suppose the table below shows the frequency with which the malware sample Regon calls the 4 API packages shown in Figure 4.1.

<table>
<thead>
<tr>
<th></th>
<th>android.view</th>
<th>java.net</th>
<th>android.app.admin</th>
<th>java.util</th>
</tr>
</thead>
<tbody>
<tr>
<td>Freq.</td>
<td>35</td>
<td>0</td>
<td>1</td>
<td>112</td>
</tr>
</tbody>
</table>

Suppose we use Equation (4.6) as our suspicion scoring method. In this case, the suspicion scores (after sorting in descending order) are given by the table:

<table>
<thead>
<tr>
<th></th>
<th>android.app.admin</th>
<th>android.view</th>
<th>java.util</th>
<th>java.net</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUS</td>
<td>1</td>
<td>0.5</td>
<td>0.5</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Suppose we now use \( W = 2 \) as the window size. In this case, there are two buckets — the first bucket has android.app.admin and android.view in it and the second has java.util and java.net in it. So the feature values for the banking trojan Regon obtained from the first bucket are: Binary Value = 1, # of API Packages = 2, FreqSum = 36, Max = 35, Median = 18, and WtSum = \( 1 \times 1 + 0.5 \times 35 = 18.5 \). The values of these features generated by the second bucket are 1, 1, 112, 112, 56, 56 respectively. Of course, this is a toy example that considers just 4 packages in the Android API instead of all 171.
Now suppose we repeat this process with Equation (4.6) and Equation (4.18). In this case, our table of suspicion ranks after sorting:

<table>
<thead>
<tr>
<th></th>
<th>java.util</th>
<th>java.net</th>
<th>android.view</th>
<th>android.app.admin</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR</td>
<td>0.1025</td>
<td>0.0811</td>
<td>0.0375</td>
<td>0.0375</td>
</tr>
</tbody>
</table>

So the feature values for banking trojan Regon obtained from the first bucket are:

Binary Value = 1, # of API Packages = 1, FreqSum = 112, Max = 112, Median = 56, and WtSum = $0.1025 \times 112 + 0.0811 \times 0 = 11.48$. Of the API calls in the second bucket, Regon has corresponding feature values 1, 2, 36, 35, 18 and $0.0375 \times (35 + 1) = 1.35$.

The above discussion about generating TSG features for the malware application Regon based on a sub-set of API packages and a part of the complete TSG illustrates how to generate TSG features for any Android application. In practice, there are 171 API packages, 24 different kinds of suspicious-score functions $\rho$, 6 methods to compute TSG features for each $\rho$. As a result, for any given app, we can generate $24 \times 6 \times \left\lceil \frac{171}{W} \right\rceil$-dimensional TSG features with a given $W$. For example, if we take $W = 10$, there would be 2,592 TSG features for any app. Under this circumstance, $W$ is a controllable parameter for security analysts. In addition, an analyst may use the same window size or different window sizes over the ranked API package list. For instance, if there are 4 sorted API packages corresponding to suspicious-scores \{0.9, 0.3, 0.29, 0.2\} given by $\rho$: (1) using unified window-size $W = 2$, they can be segmented into two buckets with \{0.9, 0.3\}, \{0.29, 0.2\}; (2) with diverse window-size $\tilde{W} = \{1, 3\}$, they can be segmented into two buckets with \{0.9\}, \{0.3, 0.29, 0.2\}. The rest procedures of deriving TSG features would remain the same.

Section 4.3

**Landmark-based Features**

We propose a set of features for Android applications based on landmarks in this section. The motivation comes from the fact that landmarks are frequently used and
referred to by human beings consciously or unconsciously in daily life. Suppose you are considering buying a house in someplace (e.g., Boston). The price of the house (in your mind) would likely depend upon several factors, one of which could be the sale price of some other houses, for example, another house with similar size and age in the same area. These houses used as reference houses are landmarks. The comparison between different houses provides you with some insight into the value of houses for the type of house of interest to you.

We would like to similarly adopt the idea of using landmarks in defining a new feature space for Android applications. Suppose there is a set $\mathcal{D}$ of Android applications (including both benign and malicious ones), where each application $i \in \mathcal{D}$ has a feature vector $\tilde{f}_i$. We first select a subset $\mathcal{L}$ of samples from $\mathcal{D}$ and set them as landmarks, then define new features for each application $i \in \mathcal{D}$ w.r.t. each landmark in $\mathcal{L}$.

4.3.1. Landmark Selection

We first want to emphasize that the applications in $\mathcal{D}$ can either be goodware or malware and whether it has a known label or not does not matter. The number of landmarks $N_L$ to be selected should be decided in advance by the analyst. We suggest keeping the size of $\mathcal{L}$ small. For example, if there are $1M$ samples in set $\mathcal{D}$, a selection of $N_L = 1K$ might be considered. In this case, it is hard for adversaries to guess the selected landmarks, let alone the landmark-based features. We propose 3 methods for selecting the set $\mathcal{L}$ of landmarks from $\mathcal{D}$.

**S1 - Random selection.** Regardless of the feature vectors $\{\tilde{f}_i\}_{i \in \mathcal{D}}$, a basic and naive landmark selection method is to randomly select $N_L$ applications from $\mathcal{D}$ as landmarks.

**S2 - clustering-based selection.** The applications in $\mathcal{D}$ are firstly clustered into $N_L$ groups. There are many well-studied algorithms for clustering and we consider 6 commonly used ones:
Advanced Features of Android Applications

(a) \textit{k}-means clustering;

(b) \textit{k}-median clustering;

(c) mean-shift clustering;

(d) DBSCAN (density-based spatial clustering of applications with noise);

(e) expectation–maximization clustering using Gaussian mixture models;

(f) agglomerative hierarchical clustering.

Each clustering algorithm has its own benefits and drawbacks. They may also perform differently due to the characteristics of the dataset $\mathcal{D}$. Without previous knowledge of the applications’ feature distribution, we are not going to claim which clustering algorithm is the optimal one. After the applications in $\mathcal{D}$ are clustered into $N_L$ groups, we then select one from each of them as the landmarks. The selection within each group can either be (1) randomized or (2) centroid-based. As a result, for landmark selection method $S_2$, there are actually $6 \times 2$ ways to realize it even without counting the variability in hyper-parameters that some of the clustering and other methods use internally.

\textbf{S3 - Max-distance heuristic selection.} Algorithm \cite{[algorithm]} shows an algorithm for selecting landmarks that are scattered across the basic feature space. Except for the set $\mathcal{D}$ and the number of landmarks $N_L$ to select, we need another input, i.e., the distance function $d(\cdot)$, which is used to evaluate the distance between two application samples regarding their feature vectors. We consider 4 kinds of distance functions:

(a) Euclidean distance;

(b) Manhattan distance;

(c) Cosine distance;
(d) Hamming distance.

This algorithm starts by randomly choosing an application from $D$ as a landmark and then iteratively adds more landmarks. In each iteration, a set of applications is randomly drawn from $D - L'$ (where $L'$ is the current set of selected landmarks) and the point that is “furthest away” (in terms of the sum/minimum/median of its distance, Line 1) from the set $L'$ is the next choice. The process ends when $N_L$ landmarks have been picked. As a result, for landmark selection method $S3$, there are actually $4 \times 3$ ways (4 distance function and 3 kinds of definitions for “furthest”) to realize it.

**Algorithm 1** Max-Distance Heuristic Landmark Selection

**Input:** Set $D$ of applications, distance function $d(\cdot)$, $N_L$

**Output:** Set $L$ of landmarks

**Initialize:** $L' \leftarrow$ Randomly select an application from $D$

**While** $|L'| < N_L$ **do**
- $R = \text{a set of applications randomly drawn from } D - L'$
- $Best = \arg\max_{r \in R} \sum_{\ell \in L} d(\ell, r)$
- $L' \leftarrow L' \cup \{Best\}$

**Return:** $L \leftarrow L'$

In total, there are $1 + 12 + 12 = 25$ ways to select the set $L$ of landmarks from the set $D$ for each $N_L$ value.

However, to further confound the adversary, the security researchers may periodically (e.g. once a week) either use a new set of landmarks or modify the landmark selection method, or both and recompute landmark-based features. By doing so, the adversary is kept guessing in the spirit of “moving target” defense[76].

### 4.3.2. Landmark-based Feature Generation

We use the selected landmarks in set $L$ to compute landmark-based features for each application sample $i$ in set $D$. First, we compute the distance $d(i, \ell)$ of sample $i$ to each $\ell \in L$, then its $N_L$-dimensional landmark-based feature vector is constructed by
using the distances \( \{d(i, \ell)\} \). As in the case of landmark selection method \( S3 \), we use 4 different distance functions.

Algorithm 2 presents the whole procedure of generating landmark-based features. We input the set \( \mathcal{D} \) of Android applications with their feature vectors \( F = \{f_i\}_{i \in \mathcal{D}} \), the number \( N_L \) of landmarks to select, the landmark selection method \( S \) and its parameters if applicable and the distance function \( d(\cdot) \). First, we generate the set \( \mathcal{L} \) of landmarks with \( S \); then iteratively compute the landmark feature vectors for each sample application \( i \in \mathcal{D} \).

**Algorithm 2 Generating LM features**

**Input:** \( \mathcal{D} \), \( F = \{f_i\}_{i \in \mathcal{D}} \) (basic feature vectors for samples in \( \mathcal{D} \)), \( N_L \) (the number of landmarks to select), landmark selection method \( S \) and distance function \( d(\cdot) \)

**Output:** landmark-based feature vector \( f_{lm_i} \) for each \( i \in \mathcal{D} \)

**Initialize:** \( \mathcal{L} \leftarrow S \)

**For** each sample application \( i \in \mathcal{D} \) \n
**For** each landmark \( \ell \in \mathcal{L} \) \n
Compute \( d(i, \ell) \);

**Return:** \( f_{lm_i} = \{d(i, \ell)\}_{\ell \in \mathcal{L}} \)

Figure 4.3 is a simple illustration of landmark features. Suppose there are 6 samples in set \( \mathcal{D} \), each with 4–dimensional API feature vector (values as the shown in Figure 4.1). 2 (Perfect Girls and Marcher) of the 6 samples are randomly selected as landmarks (with \( S1 \) in set \( \mathcal{L} \). Landmark features are generated with the Euclidean distance function. From the figure, we can see the Euclidean distance from each sample application \( i \in \mathcal{D} \) to each landmark. The landmark-based feature vector for Regon is then \( \langle 3551.33, 677.93 \rangle \), while that for Perfect Girls is \( \langle 0, 2903.66 \rangle \).

**Section 4.4 Feature clustering-based Features**

We propose another approach to generate transformed features based on feature clustering. The intuition behind this approach is that some features may share
Advanced Features of Android Applications

Figure 4.3: Illustration of landmark-based features with 6 applications, 2 landmarks and the Euclidean distance function.

similarities and demonstrate similar relationships with the label to predict. In this case, we can combine those similar features together, and thus create a smaller but perhaps more representative set of new features. The approach first clusters the set of basic features into a number of groups then derives aggregated features in each group. We call these features FC features for short.

Algorithm 3 presents the method we use to get FC features. The inputs consist of the set $D$ of all Android applications, each with their $n$-dimensional basic feature vector, the number $G$ of clusters to divide $n$ features, $Clu$ the clustering algorithm, and $\oplus$ the algorithm to aggregate features within one group. First, we extract a subset $D'$ of sample applications from $D$ and use their feature values to cluster the $n$ features into $G$ groups. Note that a subset of $D$ instead of itself is used for feature clustering. The reason is three-fold: first, the dataset $D$ might be huge and hence clustering the dataset might be very expensive; second, by using a subset of samples for clustering, we make it harder for the adversary to figure out the feature clustering result; and third, when the set $D$ is extended with more applications, the corresponding FC features of the new applications can be computed without having to re-run Algorithm 3 and re-clustering basic features. Then the FC features are computed for each application in the set $D$ w.r.t. each group of features and the associative and commutative feature

\footnote{In practice, we use API features (Section 4.1) as the basic features. This is optional and our approach can work on other selected basic features in similar procedures.}
Algorithm 3 Generating FC features

Input: $D$, $F = \{f_{ij}\}_{i \in D, 1 \leq j \leq n}$, $G$: number of clusters desired, $Clu$ a clustering algorithm, $\oplus$ associative and commutative feature aggregate algorithm

Output: $f_{ij}^{fc} = \{f_{ij}^{fc}\}_{1 \leq g \leq G}$ for each $i \in D$

Initialize: $D' \leftarrow$ take a subset of samples from $D$

$F' \leftarrow$ get the feature matrix for samples in $D'$

Cluster the $n$ basic features into $G$ groups according to column vectors $\{f_{ij}\}_{i \in D'}$ in $F'$ matrix with $Clu$

For each sample application $i \in D$ do

For each feature group $F_g$ do

$f_{ij}^{fc} = \oplus\{f_{ij} | j \in F_g\}$

Return: $f_i^{fc} = (f_{i1}^{fc}, \cdots, f_{iG}^{fc})$

4.4.1. Clustering and Feature Aggregation

Clustering: Similar to landmark selection method S2, we consider 6 clustering methods ($k$-means/median clustering, mean-shift clustering, DBSCAN, EM clustering using GM models and agglomerative hierarchical clustering) as candidates of $Clu$.

Feature aggregation: According to algorithm 3 the set $\{f_{ij} | j \in F_g\}$ of feature values for the application $i$ and the features clustered into group $F_g$ are extracted according to the basic feature vector $\vec{f}_i$. Then an associative and commutative combination operator $\oplus$ is applied on the set to aggregate them and generate a FC feature $\oplus\{f_{ij} | j \in F_g\}$. There are numerous options for $\oplus$, some of our consideration are listed below.

(a) (Product) The new feature is computed as the product of elements in the set, i.e., $f_{ig}^{fc} = \prod_{j \in F_g} f_{ij}$.

(b) (Mean) The mean value of the set of values is used as the new feature value: $f_{ig}^{fc} = \text{mean}_{j \in F_g} f_{ij}$.

(c) (Median) The mean value of the set of values is used as the new feature value: $f_{ig}^{fc} = \text{median}_{j \in F_g} f_{ij}$. 

48
(d) (Sum) The new feature is computed as the sum of elements in the set: $f_{ig}^c = \sum_{j \in \mathcal{F}_g} f_{ij}$.

(e) (WtSum) The new feature is computed as the weighted sum of elements in the set. The weight of feature $j$ is inverse proportional to the distance between the feature $j$’s vector $\{f_{ij}\}_{i \in \mathcal{D}'}$ and the centroid feature of the group $j_c$’s vector $\{f_{ijc}\}_{i \in \mathcal{D}'}$, which is denoted as $d(j, j_c)$. Thus $f_{ig}^c = \alpha \sum_{j \in \mathcal{F}_g} f_{ij} \times e^{-d(j, j_c)}$ where $\alpha$ is the parameter for normalization. Note that we try all distance measures stated for landmark selection method $S_3$ for function $d(j, j_c)$.

Note that the $G$–dimensional FC features are transformed from a $n$–dimensional basic feature vector for each Android app. Usually, we select a $G$ to be much smaller than $n$, so that the feature space dimension would decrease dramatically. For instance, if the basic feature vector has 100 elements, and $G$ is set as 8, then the FC feature vector for application $i$ would be $\{f_{i1}, \ldots, f_{i8}\}$. This FC feature vector is supposed to be highly representative. Meanwhile, they are hard to guess for adversaries, since in the process of generating FC features, security analysts have to make several key choices. These choices inject considerable uncertainty for the adversary and make it difficult for him to reverse-engineer the feature space. These choices include: (1) the selection of $\mathcal{D}' \subset \mathcal{D}$; (2) the number $G$ of clusters to generate; (3) the clustering method $Clu$ and its hyper-parameters; (4) the aggregation operator $\oplus$ and its hyper-parameters when $\oplus$ is WtSum.

For the purpose of illustration, our FC feature generation result for a simple example is also pictorially depicted in Figure 4.4. With the 6 sample applications and 4–dimensional API features shown in Figure 4.1, suppose that the 4 features are clustered into 2 groups, and $\oplus$ as the “Mean” approach (item No. 2 as stated above), we can get the FC features for each application as shown in the right-side table of Figure 4.4.
Section 4.5

Correlation Graph based Feature Transformation

Similar to the intuition behind feature value clustering-based feature transformation, we now come to another feature transformation approach that is based on a new concept called the “correlation graph”. The generated features are called CG features for short.

Unlike the FC feature transformation, the CG transformation method is based on a fully connected graph with features as its vertices. The edge between two features is weighted with the correlation of the values of those features across the space of Android applications considered. Specifically, for any two features $j_1$ and $j_2$ and a given set of Android applications, we can get two lists of feature values respectively with all applications’ feature values of $j_1$ and $j_2$. The Pearson Correlation Coefficient $c_{j_1,j_2}$ is then calculated for this pair of features $j_1$ and $j_2$. This value $c_{j_1,j_2}$ is used as the weight of the edge between features $j_1$ and $j_2$ of the correlation graph.

With the correlation graph of features, we can use community detection techniques to divide the features into a number of communities such that similar features are in the same community. We can then associate one feature value for each community.

As shown in Algorithm 4, we would first select a subset $D'$ of sample applications from $D$ and get their feature matrix $F'$ as discussed for FC features. Then the
correlation between each pair of features is computed to get the correlation graph, using which we get $G$ communities with any one of several community detection algorithms $C$. Finally, the CG features are generated for each $i \in D$ w.r.t. features in each community $C_g$ and the associative and commutative feature aggregation approach $\oplus$.

**Algorithm 4 Generating CG features**

**Input:** $D$, $F = \{f_{ij}\}_{i \in D, 1 \leq j \leq n}$, $C$ a community detection algorithm, $G$ desired number of communities, $\oplus$ an associative and commutative operator

**Output:** $F^{cg} = \{\tilde{f}_i^{cg} \mid i \in D\}$ (correlation graph based $G$-dimensional feature vectors for sample applications in $D$)

**Initialize:** $D' \leftarrow$ take a subset of samples from $D$

$F' \leftarrow$ get the feature matrix for samples in $D'$

Compute $n \times n$ edge weights of the correlation graph according to column vectors of $F'$ matrix

Get $G$ communities with the $n$ basic features according to the correlation graph and the community detection algorithm $C$

**For** each sample application $i \in D$ do

**For** each feature community $C_g$ do

$f_{ig}^{cg} = \oplus\{f_{ij} \mid j \in C_g\}$

**Return:** $\tilde{f}_i^{cg} = (f_{i1}^{cg}, \cdots, f_{iG}^{cg})$ CG feature vector for sample $i$

Similar to FC feature transformation, we have 5 different ways to define $\oplus$. As for the community detection algorithm $C$, there are multiple options, including:

(a) Minimum-cut method [77];

(b) Hierarchical clustering [78];

(c) Girvan-Newman algorithm [79];

(d) Modularity maximization [80];

(e) Statistical inference [81];

(f) Clique based methods [82].

We note that the CG feature transformation maps each app’s $n$—dimensional basic feature value to a new and much smaller $G$—dimensional space. In this process, a
number of choices are involved and they inject a great deal of uncertainty for the adversary who might be attempting to reproduce the CG features. The first choice is the size of the subset $D'$ and how to get it. Another choice is which community detection algorithm to use. A third choice would be the choice of hyper-parameters used in the community detection algorithm — even if the adversary correctly guessed the community detection algorithm used, he would still need to guess the hyper-parameters used. A final choice would be to guess the feature aggregation approach $\oplus$ within each community (besides using a single $\oplus$ operator across all communities, it is also possible to use a different $\oplus$ operator for each community).

Overall, our CG feature generation process consists of many different choices that would bring big differences in the final generated CG feature values. Adversaries will therefore have considerable difficulty in determining how it is implemented in real-world scenarios.

Figure 4.5 shows an example of correlation graph based features. Recall the samples and features presented in Figure 4.1. Suppose the 4 API features are in $G = 2$ communities as shown in the left-side figure. The CG features for each sample application while using the “Mean” feature aggregation method are listed in the right-side table.
Dataset. We create our dataset by downloading Android APKs from VirusTotal during the 2016-2017 time period that were classified as Android Banking Trojans (referred to as ABTs) by at least 5 antivirus tools. We did the same with goodware (0 detections) and with other types of Android malware (Android malware not identified as ABT, and with at least 5 antivirus detections).

Table 5.7 summarizes the composition of the dataset. The raw dataset row in this table shows the number of samples in each category that were downloaded and successfully dynamically executed on Koodous [13]. However, two samples with different file hashes may have the same feature vectors. In this case, we say that these two samples are isomorphic. Training/testing on the raw dataset would run the risk of artificially inflating prediction quality because a sample might be in the training
set, while an isomorphic copy could also be in the test set. In this case, the problem of labeling the test example would be unrealistically trivial, leading to an unjustifiable increase in prediction performance. To avoid this, we build a No-Isomorphic dataset which retained only one copy of samples that had the same feature vectors.

**Classifier training and malware detection.** Figure 5.1 shows how to train the malware classifiers. First, for the set of APK samples, we first extract basic features as described in Section 4, then we build the API package call graph and the API features (Section 4.1) are then calculated, based on which the SUS/SR (suspicious score and suspicious rank, Section 4.2.1) features, LM (landmark based, Section 4.3) features, FC (feature value clustering based, Section 4.4) features and CG (correlation graph based, Section 4.5) features are generated and stored.

With a subset of the above-mentioned features of the APK samples, we train the classifiers with different classic classifiers, including (1) Bernoulli Naive Bayes, (2) Random Forest, (3) Logistic regression, (4) SVM, (5) Nearest Neighbors, (6) Gaussian Naive Bayes, (7) Adaboost, (8) Gradient Boosting Decision Tree and (9) XGBoost.

Two additional techniques, collective classification, and late fusion are employed to further improve the classifier performance. Specifically, collective classification involves iterating the training process and late fusion combines multiple classic classifiers together.

### 5.1.1. Detection Performance

We are now ready to describe the experiments we conducted to evaluate the performance of DBank using different combinations of our novel TSG (Section 2) and traditional features. We followed the usual 10-fold cross-validation protocol. In each
Detection and Analysis of Android Malware Families

Table 5.2: Multiple metrics (AUC and F1 etc.) on Android Banking Trojans detection vs. Goodware No-Isomorphic dataset. Best Late Fusion Parameters: API + S + D: 0.344, TSG: 0.116, LM: 0.105, FC: 0, CG: 0.435.

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Best Classifier</th>
<th>AUC</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
<th>FPR</th>
<th>FNR</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Package</td>
<td>XGBoost</td>
<td>0.9862</td>
<td>0.9483</td>
<td>0.9161</td>
<td>0.9319</td>
<td>0.0517</td>
<td>0.0293</td>
</tr>
<tr>
<td>Static</td>
<td>XGBoost</td>
<td>0.9792</td>
<td>0.9780</td>
<td>0.9208</td>
<td>0.9485</td>
<td>0.0220</td>
<td>0.0275</td>
</tr>
<tr>
<td>Dynamic</td>
<td>MLP</td>
<td>0.9215</td>
<td>0.9242</td>
<td>0.6550</td>
<td>0.7667</td>
<td>0.0758</td>
<td>0.1107</td>
</tr>
<tr>
<td>Static + Dynamic</td>
<td>XGBoost</td>
<td>0.9810</td>
<td>0.9819</td>
<td>0.9208</td>
<td>0.9504</td>
<td>0.0181</td>
<td>0.0274</td>
</tr>
<tr>
<td>TSG</td>
<td>XGBoost</td>
<td>0.9750</td>
<td>0.9837</td>
<td>0.9642</td>
<td>0.9738</td>
<td>0.0163</td>
<td>0.0126</td>
</tr>
<tr>
<td>Landmark</td>
<td>KNN</td>
<td>0.5864</td>
<td>0.2826</td>
<td>0.9859</td>
<td>0.4393</td>
<td>0.7174</td>
<td>0.0419</td>
</tr>
<tr>
<td>FC</td>
<td>RF</td>
<td>0.9820</td>
<td>0.9215</td>
<td>0.8633</td>
<td>0.8915</td>
<td>0.0785</td>
<td>0.0473</td>
</tr>
<tr>
<td>CG</td>
<td>KNN</td>
<td>0.5101</td>
<td>0.2617</td>
<td>1.0000</td>
<td>0.4148</td>
<td>0.7383</td>
<td>0.0000</td>
</tr>
<tr>
<td>API + S + D + TSG</td>
<td>XGBoost</td>
<td>0.9970</td>
<td>0.9827</td>
<td>0.9661</td>
<td>0.9743</td>
<td>0.0173</td>
<td>0.0119</td>
</tr>
<tr>
<td>API + S + D + Landmark</td>
<td>XGBoost</td>
<td>0.9974</td>
<td>0.9837</td>
<td>0.9651</td>
<td>0.9743</td>
<td>0.0163</td>
<td>0.0123</td>
</tr>
<tr>
<td>API + S + D + FC</td>
<td>XGBoost</td>
<td>0.9975</td>
<td>0.9827</td>
<td>0.9642</td>
<td>0.9734</td>
<td>0.0173</td>
<td>0.0126</td>
</tr>
<tr>
<td>API + S + D + CG</td>
<td>XGBoost</td>
<td>0.9974</td>
<td>0.9827</td>
<td>0.9642</td>
<td>0.9734</td>
<td>0.0173</td>
<td>0.0126</td>
</tr>
<tr>
<td>All features</td>
<td>XGBoost</td>
<td>0.9973</td>
<td>0.9809</td>
<td>0.9670</td>
<td>0.9739</td>
<td>0.0191</td>
<td>0.0116</td>
</tr>
<tr>
<td>Best Late Fusion results</td>
<td>XGBoost</td>
<td>0.9982</td>
<td>0.9905</td>
<td>0.9736</td>
<td>0.9819</td>
<td>0.0095</td>
<td>0.0093</td>
</tr>
</tbody>
</table>

Figure 5.1: The training process setup

In order to evaluate DBank performance in distinguishing between ABTs and goodware, we tested a suite of 8 classifiers as mentioned above. We also varied the set of features provided as input to the classifiers in order to understand the impact of different feature sets on performance. We consider performance for each of these feature sets alone, as well as combinations of them.
Table 5.3: Multiple metrics (AUC and F1 etc.) on Android Banking Trojans detection vs. Other-malware No-Isomorphic dataset. Best Late Fusion Parameters: API + S + D: 0.345, TSG: 0.338, LM: 0.076, FC: 0, CG: 0.241.

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Best Classifier</th>
<th>AUC</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
<th>FPR</th>
<th>FNR</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Package</td>
<td>GBDT</td>
<td>0.9099</td>
<td>0.8470</td>
<td>0.8388</td>
<td>0.8432</td>
<td>0.1524</td>
<td>0.1603</td>
</tr>
<tr>
<td>Static</td>
<td>XGBoost</td>
<td>0.9156</td>
<td>0.8513</td>
<td>0.8401</td>
<td>0.8456</td>
<td>0.1487</td>
<td>0.1592</td>
</tr>
<tr>
<td>Dynamic</td>
<td>MLP</td>
<td>0.8394</td>
<td>0.8100</td>
<td>0.6378</td>
<td>0.7137</td>
<td>0.1900</td>
<td>0.3008</td>
</tr>
<tr>
<td>Static + Dynamic</td>
<td>XGBoost</td>
<td>0.9138</td>
<td>0.8560</td>
<td>0.8391</td>
<td>0.8475</td>
<td>0.1440</td>
<td>0.1591</td>
</tr>
<tr>
<td>API + S + D</td>
<td>XGBoost</td>
<td>0.9447</td>
<td>0.8794</td>
<td>0.8794</td>
<td>0.8794</td>
<td>0.1206</td>
<td>0.1214</td>
</tr>
<tr>
<td>TSG</td>
<td>GBDT</td>
<td>0.9117</td>
<td>0.8476</td>
<td>0.8492</td>
<td>0.8484</td>
<td>0.1524</td>
<td>0.1518</td>
</tr>
<tr>
<td>Landmark</td>
<td>XGBoost</td>
<td>0.5451</td>
<td>0.5266</td>
<td>0.8878</td>
<td>0.6611</td>
<td>0.4734</td>
<td>0.3650</td>
</tr>
<tr>
<td>FC</td>
<td>XGBoost</td>
<td>0.8499</td>
<td>0.7588</td>
<td>0.7681</td>
<td>0.7635</td>
<td>0.2412</td>
<td>0.2363</td>
</tr>
<tr>
<td>CG</td>
<td>SVM</td>
<td>0.5045</td>
<td>0.5028</td>
<td>1.0000</td>
<td>0.6692</td>
<td>0.4972</td>
<td>0.0000</td>
</tr>
<tr>
<td>API + S + D + TSG</td>
<td>XGBoost</td>
<td>0.9463</td>
<td>0.8807</td>
<td>0.8765</td>
<td>0.8786</td>
<td>0.1193</td>
<td>0.1237</td>
</tr>
<tr>
<td>API + S + D + Landmark</td>
<td>XGBoost</td>
<td>0.9444</td>
<td>0.8815</td>
<td>0.8831</td>
<td>0.8823</td>
<td>0.1185</td>
<td>0.1179</td>
</tr>
<tr>
<td>API + S + D + FC</td>
<td>XGBoost</td>
<td>0.9451</td>
<td>0.8803</td>
<td>0.8803</td>
<td>0.8803</td>
<td>0.1197</td>
<td>0.1205</td>
</tr>
<tr>
<td>API + S + D + CG</td>
<td>XGBoost</td>
<td>0.9439</td>
<td>0.8789</td>
<td>0.8756</td>
<td>0.8772</td>
<td>0.1211</td>
<td>0.1248</td>
</tr>
<tr>
<td>All features</td>
<td>XGBoost</td>
<td>0.9476</td>
<td>0.8827</td>
<td>0.8794</td>
<td>0.8810</td>
<td>0.1173</td>
<td>0.1210</td>
</tr>
<tr>
<td>Best Late Fusion results</td>
<td>XGBoost</td>
<td>0.9796</td>
<td>0.9447</td>
<td>0.9576</td>
<td>0.9507</td>
<td>0.0553</td>
<td>0.0424</td>
</tr>
</tbody>
</table>

Table 5.2 shows the AUC and FPR of using different classifiers and feature space combinations. The table considers the “No-Isomorphic” scenarios (Section 5.3.1), where feature vectors are unique. We see that XGBoost generates the best resulting irrespective of the feature types used when combining features together for classification, yielding up to 0.9819 F1 and 0.0095 FPR. In particular, when individual features are considered, our TSG performs very well with a 0.93 F1 score. The best performance of 0.9819 F1 score is achieved when a combination of all different features and late fusion is used (with basic feature’s weight as 0.344, TSG feature’s as 0.116, LM feature’s weight as 0.105, and CG feature’s weight as 0.435 in classification).

We also compare the same classifiers with different feature combinations to predict whether a malicious object belongs to either the banking trojan class (i.e., ABT) or is in the “other-malware” type category. The resulting F1 scores and FPRs are shown in Table 5.3. Again, we see that DBank with XGBoost classifier generates the best results with an F1 of 0.9507 and an FPR of 0.0553 when considering the feature combination using late fusion (with basic feature’s weight as 0.345, TSG feature’s as 0.338, LM feature’s weight as 0.076, and CG feature’s weight as 0.241 in classification), whereas our TSG features perform lower (0.8484 F1 score). Despite this slightly
lower performance of TSG in this setting, we later show that TSG features have some interesting defensive properties that make it harder for the attacker to guess the defender’s predictions (see Section 5.1.3).

Moreover, we observe that ABT vs. other-malware performance is slightly lower than what we saw when distinguishing ABTs from goodware, and reflects the fact that ABTs have more in common with other malware than they have with goodware. This is not surprising. We know that some ABTs also have a spyware component. For instance, the well-known Asacub ABT also acts as a form of spyware.

5.1.2. Key Features in Android Banking Trojan’s Detection

Key Features Distinguishing ABTs from Goodware. We also investigated the key features that distinguish ABTs from goodware in the “No-Isomorphic” case. Figure 5.2 shows 10 of the top 25 features that distinguish ABTs from goodware. Each histogram corresponds to a specific feature (e.g., number of calls to the android.view package); the X-axis reports the feature values, whereas the Y-axis reports the percentage of ABT (resp. goodware) that have a certain feature value. For example, Figure 5.2 shows that having the RECEIVE_SMS, READ_PHONE_STATE and SYSTEM_ALERT_WINDOW permission are some of the most important features distinguishing ABTs from goodware because ABTs are far more likely to have this permission than goodware. Likewise, Android apps that do not invoke any methods in the API package android.widget and/or android.view are far more likely to be ABTs than goodware.

Key Features Distinguishing ABTs from Other-malware. We also investigate the key features that distinguish ABTs from other malware in the “No-Isomorphic” case. Figure 5.3 shows 10 of the 25 top features that distinguish ABTs from other malware. We see from Figure 5.3 that the READ_SMS, GET_TASKS and CALL_PHONE permissions are very effective in distinguishing ABTs from other malware, since ABTs

1A Kaspersky report from April 2018 states that: “We encountered the Trojan-Banker.AndroidOS.Asacub family for the first time in 2015, when the first versions of the malware were detected, analyzed, and found to be more adept at spying than stealing funds.”
are far more likely to have these permissions than other-malware. These permissions are likely used by ABTs to prevent and stop any call and alert SMS from a bank notifying unusual account activity. On the other hand, the other most important features have distributions that make using them individually for predicting whether an APK is an ABT or another malware type much more challenging. This again is not surprising as functionalities and behaviors of ABTs are harder to distinguish from other malware than from goodware. We emphasize that it is the combination of these features that enables us to make good predictions, and that these features by themselves do not do that.
5.1.3. Robustness Analysis

In this section, we study the robustness of our novel TSG features against an adversary who uses machine learning using publicly available training data. Because resources such as VirusTotal are available to many people (including malicious attackers), and because large criminal networks have no trouble in gaining access to existing malware, they have access to huge numbers of Android samples. Suppose $S$ is the set of all apps available to the defender which he got through some public services (e.g., VirusTotal). In particular, $S \supseteq \{B \cup G\}$, where $B$ is the set of ABTs and $G$ is the set of goodware used by the defender for training.

We study two questions in this section:

(a) Suppose the attacker trains on a set that intersects part of $B \cup G$. How well would he infer the predictions of the defender model to craft an attack, depending on the size of this intersection? How different would the attacker feature space be?

(b) How much of $S$ do we need to use in a training set $B \cup G$ in order to ensure that we achieve and maintain high detection accuracy while deceiving an adversary who is potentially using other subsets of $S$?

The next two subsections determine the answers to these two questions.

**Robustness based on Intersection of Defender and Adversary Training Data.** We study the first question by varying the size of the intersection $\Delta \in \{10\%, 20\%, \ldots, 90\\%\}$ of the training set used by the attacker with the training set $B \cup G$ used by the defender. We call $\Delta$ the overlap ratio, which represents the percentage of data shared by both the attacker and the defender used to train their models. We define the adversary’s error rate as the percentage of attacker predictions disagreeing with the defender model (e.g., the defender predicts a sample as ABT whereas the adversary predicts it as goodware).
We randomly select 50 subsets of samples for the adversary with similar distributions as $\mathbb{B} \cup \mathbb{G}$, and measure the ratio of the adversary’s error rate, normalized by dividing it by the error rate of the baseline (consisting of static features from the APK’s Manifest and the Dynamic features) as the overlap ratio is varied (X-axis). The result is shown in Figure 5.4, both when predicting ABTs vs. goodware, as well as predicting ABTs vs. other malware. In both cases, we see that even if the adversary knows 80% of the set $\mathbb{B} \cup \mathbb{G}$ used by us, it is still the case that the error rate generated using DBank is over 3 times the error of the baseline when distinguishing ABTs from goodware, and over 1.6 times the error rate of the baseline when distinguishing ABTs from other malware.
We now also evaluate how close the feature space of the attacker is to the feature space of the defender, by varying $\Delta$ as before. But this time we study the distances between the feature vectors of defender’s samples using $B, G$ and using the samples used by the adversary. We tested our algorithm against many distance functions
including Euclidean distance, Manhattan distance, Kolomogorov-Smirnov distance, Chebyshev distance, and Cosine distance. We report results with Euclidean distance and K-S distance in the main body of the paper, while charts for the other distance metrics are reported in Appendix D (Online Supplementary Material).

Figure 5.5 and Figure 5.6 show the distances between the feature vectors generated using $B, G$ as compared to the training samples used by the adversary decreases as we vary $\Delta$. We see that the TSG features generate the biggest distances, substantially more than traditional features. In all cases, we note that as $\Delta$ increases, the distance between the feature vectors generated using $B, G$ as compared to the training samples used by the adversary decreases. This is not surprising as an increase in $\Delta$ means that the adversary more accurately guessed what we used to train on.
Figure 5.5: Euclidean distance among attacker’s and defender’s feature sets centroids. High values imply higher robustness.
In this section, we answer the second question posed at the beginning of this section. Specifically, we would like to understand whether the defender can use a subset of his training samples and change it over time to mislead the attacker. The first concern

**Accuracy-Robustness Trade-off of DBank’s TSG vs traditional features.**

Figure 5.6: Kolomogorov-Smirnov distance among attacker’s and defender’s feature sets centroids. High values imply higher robustness.

![Graph](image)

Figure 5.6: Kolomogorov-Smirnov distance among attacker’s and defender’s feature sets centroids. High values imply higher robustness.
when considering a smaller training sample is that predictive accuracy will drop. In this section, we evaluate how AUC and robustness vary when the defender uses smaller percentages $\rho$ of the training data. Each experiment also varies the overlap ratio $\Delta$, as before. The result is shown in Figure 5.7 both when predicting ABTs vs. goodware. The $x$-axis varies the percentage of defender’s training data $\rho$, while the $y$-axis reports values normalized by the value obtained with traditional features. By varying the size of the intersection overlap ratio $\Delta \in \{10\%, 20\%, \ldots, 90\%\}$ with $B \cup G$, also vary the actually used size of the total training sets with $\rho \in \{10\%, 20\%, \ldots, 90\%\}$. Again, we randomly select 50 sets of samples for the adversary with similar distributions as $B \cup G$, and measure the ratio of the prediction’s error rate, normalized by dividing it by the error rate of the baseline (consisting of API package call features, static features from the APK’s Manifest and the Dynamic features) as the overlap ratio (different non-red + lines) and the training set ratio is varied (X-axis), and the predicted AUC results by only TSG features (red + lines).

Values higher than 1 in the $y$-axis of Figure 5.7 imply that TSG-based features are better than traditional features (e.g., Manifest, Dynamic) in terms of adversary error rate. We see that the error rate generated using TSG features is always greater than the error of the baseline when distinguishing ABTs from goodware, while our TSG features yield high predictive accuracy AUCs. We also report AUC and show that it remains high in the different scenarios even when 20% or 30% of the training set is used. This allows the defender to use a moving defense surface [76] [84] by changing the specific training set over time while maintaining good predictive accuracy (AUC) performance.
Figure 5.7: AUC performance and adversary error rate of DBank’s TSG vs. traditional features using different proportions $\rho$ of the training set.
**Robustness against Fake Calls.** Another attack that can be launched against DBank is that of “fake calls”. For instance, we see from the previous discussion that a low frequency of calls to some particular Android API packages may help DBank identify a binary as an ABT. An attacker can try to evade this by making more calls to that API to avoid suspicion. A fake call to an API package adds edges to the Triadic Suspicion Graph. Suppose we define the “fake call percentage” or FCP to be the ratio of the number of fake calls in a TSG to the total number of edges in the TSG (both real and fake). Figure 5.8 shows that DBank is robust to an attack that tries to increase the FCP with the Random Forest classifier used in DBank showing AUCs close to 1.

![Figure 5.8: Impact of Fake Call Attack on DBank’s Performance](image)

**Section 5.2**

**A Data-Driven Characterization of Modern Android Spyware**

**Dataset.** We download 5000 spyware, 5000 goodware and 5000 other malware samples from VirusTotal [85], a service that scans suspicious files and URLs submitted by
users to be tested against multiple commercial AV systems. All samples in our dataset were first submitted to VirusTotal between July 1 2016 and July 1 2017. This choice is also motivated by [86] which empirically determines that VirusTotal’s AV detections become stable one year after initial submission.

We use Symantec threat descriptions [87] to identify spyware families. In particular, we consider an Android malware family to be a spyware family if Symantec indicates that the malware family steals information such as location, browsing history, credentials, contacts, or photos. In total, we obtain 54 spyware family names. It is worth noting that some spyware families may also have some overlapping behaviors typical of other malware categories (e.g., sending premium SMSs, banking trojans). We use the VirusTotal Intelligence API to download Android APKs belonging to our list of spyware families that were detected as malicious by at least 10 antivirus engines [86].

We also use the VirusTotal Intelligence API to collect malware samples that are not spyware; we refer to this category as other-malware. Finally, we collect goodware by querying the VirusTotal Intelligence API for samples first found in the wild between July 2016 and July 2017 which were not labeled as malware by any of the 63 VirusTotal antivirus engines (i.e., detection rate was 0%).

5.2.1. Supervised Learning Algorithms and Ensemble Late Fusion (ELF)

Traditional Classifiers. We consider six widely used traditional classifiers [88]: Random Forest (RF), Decision Tree (DT), Support Vector Machine (SVM), K-Nearest Neighbors (KNN), Naïve Bayes (NB), Logistic Regression (LR). Traditional classification algorithms take as their input a set of (feature vector, class) pairs. For

---

2We also tried to download the following families in [87] from VirusTotal [85] but no samples were present (for the period July 2016 - July 2017): Spywaller, Sockrat, Bossefiv, Alienspy, Acсstealer, Gomal, Fitikser, Ballonpop, Repane, Dupvert, Sberick, Simhosy, ZertSecurity, Teelog, Yatoot.

3In the query submitted to VirusTotal, we look for Android APKs that have been detected as malicious by at least 10 antivirus engines, and that do not belong to our list of spyware families and do not contain “spyware” keywords. As in related work [36], we also decide to exclude adware from this other-malware category because they usually represent applications that annoy users with an excessive number of ads, as opposed to compromise user devices.
instance, for each Android app, we extract a feature as described in Section 3 and the class is either 1 ("spyware") or 0 ("goodware" or "other-malware", depending on the experiment we are considering). Traditional classification algorithms try to find different equational forms that separate one class from the others. For instance, linear Support Vector Machines (SVM) try to draw a hyperplane that splits the feature vector space into two classes. SVM may use different equational forms: linear SVM uses a straight hyperplane, while kernel SVMs use diverse shapes. Other traditional classifiers (e.g. Decision Tree) use a set of generalized (to higher dimensions) rectangles to state that if a feature vector is within one of these generalized rectangles then it is most likely spyware; and in the other class (either goodware or other malware, depending on the problem we study) when it is not in any such rectangle. Hence, different classifiers try to split the space into a “spyware” part and a second part (either goodware or other-malware). As the assumptions made by some classifiers may be inconsistent with the actual data: some traditional classifiers may perform well, while others may perform poorly. One of the major goals of machine learning is to identify the right classifier for any given data set. We, therefore, use multiple traditional classifiers to see which one is best at distinguishing spyware vs. goodware, and spyware vs. other malware types.

**Deep Learning Classifiers.** We additionally consider four representative deep learning classifiers which have achieved outstanding results in many ML tasks: Multi-Layer Perceptrons (MLP), Bernoulli Restricted Boltzmann Machines (BRBM), Convolutional Neural Networks (CNN), and the Wide & Deep [89] DL architecture recently proposed by Google. We do not consider sequence-based classifiers as the dynamic analysis logs are a collection of events without timestamps, and because our goal is to consider static and dynamic features together (see Section 3). We performed extensive hyper-parameter tuning to achieve the best results with deep learning methods (despite
the fact that they are less accurate than shallow learning methods, probably due to the limited data available).

**Ensemble Late Fusion (ELF).** Our ELF architecture shown in Figure 5.9 combines the results of traditional and deep classifiers into a unified prediction. The first part of ELF follows a traditional cycle and is shown in the top part of Figure 5.9. From a set of Android APKs, we extract a set of static and dynamic features and then perform a supervised classification step using a set of classifiers. Instead of using the binary prediction generated, ELF, uses the probability returned by each classifier that a particular Android app belongs to the class 1 (spyware) vs. the class 0 (either goodware or other malware). ELF then computes a weighted sum of these probabilities by assigning a weight to each classifier such that the weights add up to one. In order to assign these weights, ELF performs a grid search to identify near-optimal weights without looking at the test set.

To describe ELF more formally, let us consider a binary classification task where an object $x_j$ can have a predicted label $\hat{y}_j = 0$ (goodware or other malware), and $\hat{y}_j = 1$
Detection and Analysis of Android Malware Families

(spyware). Each classifier outputs a probability of belonging to class 1, \( p_j \in [0.0, 1.0] \), where for a given object \( x_j \), if \( p_j > 0.5 \) then the predicted label \( \hat{y}_j = 1 \), else \( \hat{y}_j = 0 \). Since we are considering several supervised classifiers, the \( i \)-th classifier outputs a certain probability \( p^i_j \) that an object \( x_j \) belongs to class 1. The ensemble algorithm computes a Late Fusion Score (LFS) as the weighted sum of the probabilities of all the classifiers:

\[
LFS_j = \sum_{i \in \text{classifiers}} w_i p^i_j
\]  

(5.1)

where \( w_i \) is the weight (relevance) of the \( i \)-th classifier in the decision, and \( \sum_i w_i = 1 \). Then, if \( LFS_j > 0.5 \) an object \( x_j \) is assigned label \( \hat{y}_j = 1 \), otherwise \( \hat{y}_j = 0 \).

**Training ELF weights.** We now describe how we identify the best weights for ELF to use. We first split the dataset into two parts: ELF training set and ELF testing set. The identification of the weights relies exclusively on the training set, on which we perform 10-fold cross-validation with all possible weights combinations of the different classifiers; the performance on the validation is determined according to the weighted score described previously in Equation 5.1. The optimal ELF weights correspond to the ones that obtain the highest performance on the validation set, for all the 10-folds. Note that the testing set is never involved in the weight-training process. This weight training procedure is repeated in a 10-fold CV fashion for different train/test splits. The final ELF results that we report are the 10-fold average performance obtained by using the best weights found with this procedure.

5.2.2. Evaluation

We evaluate the accuracy of the different classifiers described earlier using 10-fold cross-validation. Identifying the best performing classifier is a necessary step to determine the features that best distinguish spyware from goodware and from other malware.
Table 5.4: Performance of classifiers in spyware vs goodware (average with 10-fold cross-validation). The best traditional classifier is RF. The best overall classifier is ELF with weights $w_{RF} = 0.711$, $w_{Wide\&Deep} = 0.222$, $w_{BRBM} = 0.045$, and $w_{SVM} = 0.022$. The p-values are computed via a t-test comparing $F_1$-Scores. We also use standard statistical notation $***$, $**$, $*$ ($*** < 0.01; ** < 0.05; * < 0.1$).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>$F_1$</th>
<th>P</th>
<th>R</th>
<th>AUC</th>
<th>FPR</th>
<th>FNR</th>
<th>p-val vs. RF</th>
<th>p-val vs. ELF</th>
</tr>
</thead>
<tbody>
<tr>
<td>RF</td>
<td>0.976</td>
<td>0.990</td>
<td>0.963</td>
<td>0.977</td>
<td>0.98%</td>
<td>3.70%</td>
<td>N/A</td>
<td>$*** (4.309 \cdot 10^{-53})$</td>
</tr>
<tr>
<td>DT</td>
<td>0.971</td>
<td>0.970</td>
<td>0.973</td>
<td>0.971</td>
<td>3.08%</td>
<td>2.70%</td>
<td>$*** (1.302 \cdot 10^{-39})$</td>
<td>$*** (1.028 \cdot 10^{-122})$</td>
</tr>
<tr>
<td>KNN</td>
<td>0.869</td>
<td>0.868</td>
<td>0.870</td>
<td>0.868</td>
<td>13.36%</td>
<td>13.01%</td>
<td>$*** (8.897 \cdot 10^{-282})$</td>
<td>$*** (1.158 \cdot 10^{-287})$</td>
</tr>
<tr>
<td>NB</td>
<td>0.782</td>
<td>0.663</td>
<td>0.954</td>
<td>0.732</td>
<td>48.95%</td>
<td>4.80%</td>
<td>$*** (0)$</td>
<td>$*** (0)$</td>
</tr>
<tr>
<td>LR</td>
<td>0.704</td>
<td>0.774</td>
<td>0.645</td>
<td>0.727</td>
<td>19.08%</td>
<td>35.45%</td>
<td>$*** (0)$</td>
<td>$*** (0)$</td>
</tr>
<tr>
<td>SVM</td>
<td>0.423</td>
<td>1.000</td>
<td>0.269</td>
<td>0.635</td>
<td>0.00%</td>
<td>73.09%</td>
<td>$*** (0)$</td>
<td>$*** (0)$</td>
</tr>
<tr>
<td>MLP</td>
<td>0.674</td>
<td>0.907</td>
<td>0.537</td>
<td>0.741</td>
<td>5.59%</td>
<td>46.30%</td>
<td>$*** (0)$</td>
<td>$*** (0)$</td>
</tr>
<tr>
<td>BRBM</td>
<td>0.701</td>
<td>0.918</td>
<td>0.568</td>
<td>0.758</td>
<td>5.15%</td>
<td>43.18%</td>
<td>$*** (0)$</td>
<td>$*** (0)$</td>
</tr>
<tr>
<td>CNN</td>
<td>0.623</td>
<td>0.590</td>
<td>0.783</td>
<td>0.587</td>
<td>62.90%</td>
<td>21.73%</td>
<td>$*** (3.157 \cdot 10^{-169})$</td>
<td>$*** (1.568 \cdot 10^{-165})$</td>
</tr>
<tr>
<td>W&amp;D [89]</td>
<td>0.671</td>
<td>0.883</td>
<td>0.548</td>
<td>0.742</td>
<td>8.21%</td>
<td>45.19%</td>
<td>$*** (6.993 \cdot 10^{-319})$</td>
<td>$*** (3.859 \cdot 10^{-318})$</td>
</tr>
<tr>
<td>ELF</td>
<td>0.982</td>
<td>0.988</td>
<td>0.977</td>
<td>0.982</td>
<td>1.23%</td>
<td>2.34%</td>
<td>$*** (4.309 \cdot 10^{-53})$</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Metrics.** We use the following traditional ML performance metrics: $F_1$-Score, Precision, Recall, AUC (AUROC), FPR (False Positive Rate), and FNR (False Negative Rate) for a complete view of the performance of every approach. We also report $p$-values obtained via a Student’s t-test in order to show statistical significance. Readers will recall that a result is statistically significant when the p-value is less than 0.05 [88]. Moreover, we recall that: the lower the p-value, the higher the statistical significance.

**Spyware vs. Goodware.** Table 5.4 shows the performance of different traditional classifiers in separating spyware from goodware.

**Traditional and Deep Classifiers Performance.** Of the individual algorithms, Random Forest and Decision Trees achieve the best accuracy: both achieve an $F$-Score and AUC of 0.97, with RF performing slightly better. Linear SVM is the worst, possibly because the feature space is not linearly separable—in contrast, KNN, DT and RF can draw non-linear decision boundaries. Despite extensive hyper-parameter
tuning, the deep learning algorithms do not achieve good performance, probably to the limited amount of training data. Decision Trees achieve the lowest false negative rate (2.70%), which suggests that it largely avoids mislabeling spyware as goodware, and a relatively low false positive rate (3.08%). Random Forest achieves the lowest false positive rate (0.98%) while keeping the false negative rate also relatively low (3.70%). Keeping false positives low is important in order to prevent warnings from getting ignored. In summary, Decision Tree is the preferred classifier if low false negatives are the priority and Random Forest is the best classifier if low false positives are the priority.

**ELF Performance.** We trained ELF on the training data by trying all possible weights \( w_i \) in steps of 0.05, so that the following constraint is preserved: \( \sum_i w_i = 1 \). Because we have six shallow traditional classifiers and four deep learning classifiers in the ensemble, we want to find a vector of weights \( (w_1, \ldots, w_{10}) \) such that assigning these weights to each classifier’s predictions maximizes \( F_1 \)-Score and \( AUC \). After performing tuning of ELF weights with the process described in Section ??, we identify the following optimal positive weights: \( w_{RF} = 0.711, w_{Wide & Deep} = 0.222, w_{BRBM} = 0.045, \) and \( w_{SVM} = 0.022 \) (all other weights are 0). These weights represent the relative importance of each decision algorithm in the classification, suggesting that ELF identifies Random Forest as the most important base predictor, Wide & Deep \([89]\) as the second, followed by BRBM and SVM. The last row of Table 5.4 shows the performance of ELF. We observe that ELF improves all performance metrics, and minimizes the trade-off between FPR and FNR. It is also relevant to observe that while the improvement may seem minor (e.g., \( F_1 \)-Score of 0.982 for ELF vs. 0.976 for RF), it is actually pretty large as the maximal performance can only go up to 1.

We also used the t-test to compute the p-values in Table 5.4 comparing the \( F_1 \) score of ELF against DT and RF and ELF using 10-fold CV repeated 30 times. As all the p-values are far below 0.01, the finding that ELF is superior to both RF and DT
Detection and Analysis of Android Malware Families

is statistically significant.

Table 5.5: Performance of classifiers in spyware vs other-malware (average with 10-fold cross-validation). The best traditional classifier is RF, which is outperformed by ELF with positive weights $w_{RF} = 0.967$ and $w_{BRBM} = 0.033$. The p-values are computed through the t-test performed against the $F_1$-Score results and, in addition to the raw p-values, we also use the statistical confidence notation ***, **, * (** < 0.01; ** < 0.05; * < 0.1).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>$F_1$</th>
<th>P</th>
<th>R AUC</th>
<th>FPR</th>
<th>FNR</th>
<th>p-val vs. RF</th>
<th>p-val vs. ELF</th>
</tr>
</thead>
<tbody>
<tr>
<td>RF</td>
<td>0.955</td>
<td>0.963</td>
<td>0.945</td>
<td>0.959</td>
<td>2.97%</td>
<td>5.35%</td>
<td>N/A</td>
</tr>
<tr>
<td>DT</td>
<td>0.949</td>
<td>0.945</td>
<td>0.953</td>
<td>0.954</td>
<td>4.60%</td>
<td>4.65%</td>
<td>*** (2.798 · 10^-47)</td>
</tr>
<tr>
<td>KNN</td>
<td>0.784</td>
<td>0.778</td>
<td>0.790</td>
<td>0.802</td>
<td>18.61%</td>
<td>21.04%</td>
<td>*** (0)</td>
</tr>
<tr>
<td>LR</td>
<td>0.707</td>
<td>0.771</td>
<td>0.655</td>
<td>0.747</td>
<td>16.07%</td>
<td>34.48%</td>
<td>*** (0)</td>
</tr>
<tr>
<td>NB</td>
<td>0.416</td>
<td>0.745</td>
<td>0.290</td>
<td>0.604</td>
<td>8.23%</td>
<td>71.04%</td>
<td>*** (0)</td>
</tr>
<tr>
<td>SVM</td>
<td>0.445</td>
<td>0.996</td>
<td>0.287</td>
<td>0.643</td>
<td>0.09%</td>
<td>71.28%</td>
<td>*** (0)</td>
</tr>
<tr>
<td>MLP</td>
<td>0.120</td>
<td>0.645</td>
<td>0.067</td>
<td>0.518</td>
<td>3.14%</td>
<td>93.35%</td>
<td>*** (0)</td>
</tr>
<tr>
<td>BRBM</td>
<td>0.717</td>
<td>0.662</td>
<td>0.783</td>
<td>0.726</td>
<td>33.15%</td>
<td>21.68%</td>
<td>*** (0)</td>
</tr>
<tr>
<td>CNN</td>
<td>0.028</td>
<td>0.138</td>
<td>0.015</td>
<td>0.501</td>
<td>0.79%</td>
<td>98.45%</td>
<td>*** (0)</td>
</tr>
<tr>
<td>W&amp;D [89]</td>
<td>0.074</td>
<td>0.653</td>
<td>0.040</td>
<td>0.418</td>
<td>1.92%</td>
<td>95.99%</td>
<td>*** (0)</td>
</tr>
</tbody>
</table>

**Spyware vs. Other Malware. Performance of Traditional and Deep Classifiers.** Table 5.5 reports the performance of different traditional classifiers in separating spyware from other malware using 10-fold cross-validation. RF achieves the best performance: 0.955 F-Score and 0.959 AUC, with 2.97% false positive rate (FPR) and 5.35% false negative rate (FNR). Here false negatives mean that spyware is mislabeled as other malware which could potentially delay analysts seeking to develop signatures and patches.

Though our ability to separate spyware from other forms of malware is quite high, the results in distinguishing spyware from other malware (shown in Table 5.5) are slightly lower than those for distinguishing spyware vs goodware (Table 5.4). This is because spyware is more similar to other malware than to goodware, and hence the classifier finds this task more challenging. For instance, some banking trojans (e.g., Acecard) may behave like spyware in order to more effectively carry out banking...
Performance of ELF. We now consider the performance of ELF. As in the previous subsection, we use 10-fold cross-validation on the training data to find the weights $w_i$ of the ensemble while requiring that $\sum_i w_i = 1$. The weights that maximize $F_1$-Score and $AUC$ are: $w_{RF} = 0.967$ and $w_{BRBM} = 0.033$ (all other classifiers weights are 0). Note that these weights are different from the case in which we tried to separate spyware from goodware. We report the performance of ELF in the last row of Table 5.5. We observe that by combining the algorithms, ELF improves all performance metrics, and minimizes the trade-off between FPR and FNR. Again, while the improvement may seem minor, performance is already very high. As before, Table 5.4 shows the result of performing a t-test comparing ELF’s $F_1$-score and AUC to those of the best performing individual classifiers. As all the p-values are substantially below 0.01, ELF’s superior performance is not due to chance and has statistical significance.

5.2.3. Distinguishing Characteristics of Spyware

The principal goal of this paper is to identify features that best separate spyware from goodware and from other malware. We do this via the Mean Decreased Impurity (MDI) metric. MDI is a traditional feature selection method used by Decision Trees in the Random Forest algorithm which progressively splits individual features in order to separate the data more effectively. Minimal impurity is achieved when all the objects belong to a single label. For example, the presence or absence of certain permission (e.g., \texttt{READ\_PHONE\_STATE}) may be used by DTs to split feature vectors into two separate groups to decrease impurity. We consider the traditional version with the Gini definition of impurity. We use MDI scores to build feature-value histograms that show the distributions of feature values for the two classes (e.g. spyware vs. goodware or spyware vs. other malware).
Table 5.6: Features distinguishing spyware from goodware and from other malware

<table>
<thead>
<tr>
<th>Category</th>
<th>Static features</th>
<th>Dynamic features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spyware</td>
<td>-SEND_SMS&lt;br&gt;-RECEIVE_SMS&lt;br&gt;-READ_PHONE_STATE&lt;br&gt;-READ_SMS&lt;br&gt;-RECEIVE_BOOT_COMPLETED&lt;br&gt;-WRITE_SMS&lt;br&gt;-GET_TASKS&lt;br&gt;-CALL_PHONE&lt;br&gt;-CHANGE_NETWORK_STATE&lt;br&gt;-more dangerous perms.&lt;br&gt;-more std permissions&lt;br&gt;-specific authors&lt;br&gt;-PROCESS_OUTGOING_CALL&lt;br&gt;-ACCESS_WIFI_STATE&lt;br&gt;-WRITE_CONTACTS&lt;br&gt;-ACCESS_FINE_LOCATION&lt;br&gt;-ACCESS_COARSE_LOCATION&lt;br&gt;-RECORD_AUDIO&lt;br&gt;-WRITE_CONTACTS</td>
<td></td>
</tr>
<tr>
<td>Goodware</td>
<td>-greater filesize&lt;br&gt;-more components</td>
<td>-more read /dev/&lt;br&gt;-more crypto operations</td>
</tr>
<tr>
<td>Other Malware</td>
<td>-SYSTEM_ALERT_WINDOW&lt;br&gt;-MOUNT_UNMOUNT_FILESYSTEMS&lt;br&gt;-WRITE_EXTERNAL_STORAGE</td>
<td>-write .dex classes&lt;br&gt;-load .dex classes&lt;br&gt;-servicestart AdminService</td>
</tr>
</tbody>
</table>

**Spyware vs. Goodware and Other Malware.** Table 5.6 shows a summary of the features which best separate spyware from goodware and from other malware (the Appendix reports the full feature histograms). Each group-row in Table 5.6 corresponds to a different category (i.e., spyware, goodware, other malware), and the presence of a static (resp. dynamic) features implies a prevalence of that feature in that category and its absence (or lower prevalence) from the other ones. We can observe that:

- The feature **SEND_SMS** is primarily prevalent in spyware and less present in goodware and other malware.

- Spyware apps tend to have a smaller filesize and fewer Android components than goodware. This may indicate that most spyware is not repackaged versions
of goodware apps. They could be developed as ad-hoc malware to steal user information or might be repackaged versions of small apps.

- Static features (specifical requests for Android permissions) play an important role in separating spyware from other categories. The relevance of “permission” features is related to the fact that Android applications need to ask for permission in advance in order to access certain resources (e.g., software or hardware). Some of the most relevant permissions are related to: accessing sensitive information (e.g., `READ_PHONE_STATE`, `GET_TASKS`) and sending information to attackers (e.g., `SEND_SMS`, `CHANGE_NETWORK_STATE`).

- One of the main permissions that other types of malware request is the `SYSTEM_ALERT_WINDOW` permission, which allows displaying windows on top of the screen. For example, this permission can be used by mobile ransomware in order to block access to devices or to use hidden overlays for privilege escalation [30]. This permission also has legitimate uses such as showing Facebook chat notifications. Spyware requires this permission much less frequently, but other malware often uses this permission to keep windows hidden from the user.

- Finally, it is interesting to observe that none of the Android spyware samples starts an Android Service named `AdminService`, whereas other types of malware do—likely to start stealthy background processes with a name that looks legitimate to users.
Section 5.3

Android Malware Detection via (Somewhat) Robust Irreversible Feature Transformations

5.3.1. The FARM Dataset

In this section, we briefly introduce the FARM dataset which consists of a mix of Android goodware, rooting malware, spyware, banking trojans, and other malware. For a sample to be tagged in one of these malware categories, we required that there be at least 2 reports on Koodous\footnote{https://koodous.com/} confirming this status. Table 5.7 summarizes the statistics of the FARM dataset.

Table 5.7: Dataset description

<table>
<thead>
<tr>
<th>Number of APKs</th>
<th>Isomorphic</th>
<th>No-Isomorphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goodware</td>
<td>3535</td>
<td>2999</td>
</tr>
<tr>
<td>Rooting Malware</td>
<td>1829</td>
<td>444</td>
</tr>
<tr>
<td>Banking Trojans</td>
<td>7107</td>
<td>1061</td>
</tr>
<tr>
<td>Spyware</td>
<td>3247</td>
<td>841</td>
</tr>
<tr>
<td>Other-Malware (Not Rooting)</td>
<td>4596</td>
<td>2081</td>
</tr>
<tr>
<td>Other-Malware (Not Banking Trojans)</td>
<td>3973</td>
<td>1806</td>
</tr>
<tr>
<td>Other-Malware (Not Spyware)</td>
<td>4382</td>
<td>1922</td>
</tr>
</tbody>
</table>

5.3.2. Experimental Evaluation

In this section, we describe the results of the experiments that we have designed to evaluate the performance of FARM with different feature combinations. Our experimental evaluation includes four parts: (1) Distinguishing each of the 3 types Android malware (banking trojans, rooting malware, spyware) from goodware on both the Isomorphic and No-Isomorphic datasets; (2) Distinguishing each of the 3 types Android malware (banking trojans, rooting malware, spyware) from other types of malware on both the Isomorphic and No-Isomorphic datasets; (3) Evaluating
Table 5.8: Multiple metrics (AUC and F1 etc.) on Android Malware detection vs. Goodware / Other-malware No-Isomorphic dataset.

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Best Settings</th>
<th>N</th>
<th>Distance</th>
<th>Classifier</th>
<th>AUC</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
<th>FPR</th>
<th>FNR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goodware vs. Rooting</td>
<td>SD + API</td>
<td>-</td>
<td>RF</td>
<td>RF</td>
<td>0.9845</td>
<td>0.9908</td>
<td>0.9192</td>
<td>0.9535</td>
<td>0.0092</td>
<td>0.0027</td>
</tr>
<tr>
<td>Goodware vs. Banking Trojan</td>
<td>SD + API</td>
<td>-</td>
<td>RF</td>
<td>RF</td>
<td>0.9967</td>
<td>0.9903</td>
<td>0.9696</td>
<td>0.9788</td>
<td>0.0097</td>
<td>0.0321</td>
</tr>
<tr>
<td>Goodware vs. Spysware</td>
<td>SD + API</td>
<td>-</td>
<td>RF</td>
<td>RF</td>
<td>0.9976</td>
<td>0.9920</td>
<td>0.9734</td>
<td>0.9826</td>
<td>0.0080</td>
<td>0.0282</td>
</tr>
<tr>
<td>Other-malware vs. Rooting</td>
<td>SD + API</td>
<td>-</td>
<td>RF</td>
<td>RF</td>
<td>0.9740</td>
<td>0.9181</td>
<td>0.9525</td>
<td>0.9349</td>
<td>0.0819</td>
<td>0.0537</td>
</tr>
<tr>
<td>Other-malware vs. Banking Trojan</td>
<td>SD + API</td>
<td>-</td>
<td>RF</td>
<td>RF</td>
<td>0.9671</td>
<td>0.9279</td>
<td>0.9115</td>
<td>0.9196</td>
<td>0.0721</td>
<td>0.1000</td>
</tr>
<tr>
<td>Other-malware vs. Spysware</td>
<td>SD + API</td>
<td>-</td>
<td>RF</td>
<td>RF</td>
<td>0.9876</td>
<td>0.9759</td>
<td>0.9819</td>
<td>0.9788</td>
<td>0.0241</td>
<td>0.0561</td>
</tr>
</tbody>
</table>

In addition, FARM discovered two new rooting malware samples - a fact that was not previously known to any of the 61 anti-virus engines on VirusTotal. As (1) and (2) involve 12 experiments in all, we present a sample in the main body of the paper. Readers may find more details at [part of the paper](https://drive.google.com/open?id=14ZQyFtsu6exZhoav4z-1aDZXWrmMNjy).

5.3.3. No Adversarial Attack Case

Tables 5.8 and 5.9 summarize the results of experiments on 5 settings (described [here](https://drive.google.com/open?id=14ZQyFtsu6exZhoav4z-1aDZXWrmMNjy)).

Late Fusion. The predicted probabilities $p_i$ of the $M = 8$ classifiers $C_i$, $i = 1, \ldots, 8$ are linearly combined by FARM as $p = \sum_{i=1}^{M} \gamma_i p_i$, where $\sum \gamma_i = 1$. We find the best value of the $\gamma_i$s by doing a grid search and optimizing performance on the training set.
Table 5.9: Multiple metrics (AUC and F1 etc.) on Android Malware detection vs. Goodware / Other-malware Isomorphic dataset.

<table>
<thead>
<tr>
<th>Datasets vs.</th>
<th>Best Settings</th>
<th>N</th>
<th>Distance</th>
<th>Classifier</th>
<th>AUC</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
<th>FPR</th>
<th>FNR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goodware vs. Rooting</td>
<td>SD + API</td>
<td>-</td>
<td>-</td>
<td>RF</td>
<td>0.9943</td>
<td>0.9932</td>
<td>0.8776</td>
<td>0.9312</td>
<td>0.0007</td>
<td>0.0231</td>
</tr>
<tr>
<td>Goodware vs. Banking Trojan</td>
<td>SD + API</td>
<td>-</td>
<td>-</td>
<td>RF</td>
<td>0.9974</td>
<td>0.9869</td>
<td>0.9306</td>
<td>0.8577</td>
<td>0.0000</td>
<td>0.0021</td>
</tr>
<tr>
<td>Goodware vs. Spyware</td>
<td>SD + API</td>
<td>-</td>
<td>-</td>
<td>RF</td>
<td>0.9949</td>
<td>0.9697</td>
<td>0.9697</td>
<td>0.9697</td>
<td>0.0003</td>
<td>0.0033</td>
</tr>
<tr>
<td>Other-malware vs. Rooting</td>
<td>SD + API</td>
<td>-</td>
<td>-</td>
<td>RF</td>
<td>0.9937</td>
<td>0.9817</td>
<td>0.8636</td>
<td>0.9181</td>
<td>0.0018</td>
<td>0.0261</td>
</tr>
<tr>
<td>Other-malware vs. Banking Trojan</td>
<td>SD + API</td>
<td>-</td>
<td>-</td>
<td>RF</td>
<td>0.9498</td>
<td>0.8835</td>
<td>0.8949</td>
<td>0.8887</td>
<td>0.1165</td>
<td>0.1070</td>
</tr>
<tr>
<td>Other-malware vs. Spyware</td>
<td>SD + API</td>
<td>-</td>
<td>-</td>
<td>RF</td>
<td>0.9922</td>
<td>0.9582</td>
<td>0.9733</td>
<td>0.9657</td>
<td>0.0418</td>
<td>0.0293</td>
</tr>
<tr>
<td>SET 1 Baseline</td>
<td>LM</td>
<td>-</td>
<td>-</td>
<td>RF</td>
<td>0.9836</td>
<td>1.0000</td>
<td>0.9534</td>
<td>0.9761</td>
<td>0.0000</td>
<td>0.0060</td>
</tr>
<tr>
<td>Goodware vs. Rooting</td>
<td>LF(LM-Cluster, SD, API)</td>
<td>12</td>
<td>Hamming</td>
<td>RF</td>
<td>0.9983</td>
<td>1.0000</td>
<td>0.9594</td>
<td>0.9761</td>
<td>0.0000</td>
<td>0.0060</td>
</tr>
<tr>
<td>Goodware vs. Banking Trojan</td>
<td>LF(LM-Cluster, SD, API)</td>
<td>30</td>
<td>Hamming</td>
<td>GBDT</td>
<td>0.9952</td>
<td>1.0000</td>
<td>0.9565</td>
<td>0.9778</td>
<td>0.0000</td>
<td>0.0068</td>
</tr>
<tr>
<td>Goodware vs. Spyware</td>
<td>LF(LM-Cluster, SD, API)</td>
<td>24</td>
<td>Hamming</td>
<td>RF</td>
<td>0.9972</td>
<td>0.9903</td>
<td>0.9697</td>
<td>0.9798</td>
<td>0.0007</td>
<td>0.0320</td>
</tr>
<tr>
<td>Other-malware vs. Rooting</td>
<td>LF(LM-Cluster, SD, API)</td>
<td>30</td>
<td>Hamming</td>
<td>RF</td>
<td>0.9826</td>
<td>0.9241</td>
<td>0.9521</td>
<td>0.9378</td>
<td>0.0759</td>
<td>0.0534</td>
</tr>
<tr>
<td>Other-malware vs. Banking Trojan</td>
<td>LF(LM-Cluster, SD, API)</td>
<td>18</td>
<td>Hamming</td>
<td>GBDT</td>
<td>0.9614</td>
<td>0.9352</td>
<td>0.9619</td>
<td>0.9484</td>
<td>0.0664</td>
<td>0.0388</td>
</tr>
<tr>
<td>Other-malware vs. Spyware</td>
<td>LF(LM-Cluster, SD, API)</td>
<td>21</td>
<td>Hamming</td>
<td>RF</td>
<td>0.9974</td>
<td>0.9915</td>
<td>0.9738</td>
<td>0.9826</td>
<td>0.0085</td>
<td>0.0277</td>
</tr>
<tr>
<td>SET 2</td>
<td>FC</td>
<td>-</td>
<td>-</td>
<td>RF</td>
<td>0.9990</td>
<td>1.0000</td>
<td>0.9705</td>
<td>0.9850</td>
<td>0.0000</td>
<td>0.0030</td>
</tr>
<tr>
<td>Goodware vs. Rooting</td>
<td>LF(FC, SD, API)</td>
<td>27</td>
<td>-</td>
<td>RF</td>
<td>0.9990</td>
<td>1.0000</td>
<td>0.9705</td>
<td>0.9850</td>
<td>0.0000</td>
<td>0.0030</td>
</tr>
<tr>
<td>Goodware vs. Banking Trojan</td>
<td>LF(FC, SD, API)</td>
<td>30</td>
<td>-</td>
<td>XGB</td>
<td>0.9949</td>
<td>1.0000</td>
<td>0.9444</td>
<td>0.9714</td>
<td>0.0000</td>
<td>0.0066</td>
</tr>
<tr>
<td>Goodware vs. Spyware</td>
<td>LF(FC, SD, API)</td>
<td>21</td>
<td>-</td>
<td>RF</td>
<td>0.9969</td>
<td>0.9910</td>
<td>0.9709</td>
<td>0.9808</td>
<td>0.0090</td>
<td>0.0307</td>
</tr>
<tr>
<td>Other-malware vs. Rooting</td>
<td>LF(FC, SD, API)</td>
<td>21</td>
<td>-</td>
<td>RF</td>
<td>0.9696</td>
<td>0.9090</td>
<td>0.9331</td>
<td>0.9208</td>
<td>0.0910</td>
<td>0.0743</td>
</tr>
<tr>
<td>Other-malware vs. Banking Trojan</td>
<td>LF(FC, SD, API)</td>
<td>30</td>
<td>-</td>
<td>XGB</td>
<td>0.9669</td>
<td>0.8991</td>
<td>0.9515</td>
<td>0.9245</td>
<td>0.0109</td>
<td>0.0409</td>
</tr>
<tr>
<td>Other-malware vs. Spyware</td>
<td>LF(FC, SD, API)</td>
<td>24</td>
<td>-</td>
<td>RF</td>
<td>0.9978</td>
<td>0.9856</td>
<td>0.9865</td>
<td>0.9860</td>
<td>0.0144</td>
<td>0.0147</td>
</tr>
<tr>
<td>SET 3</td>
<td>CG</td>
<td>-</td>
<td>-</td>
<td>RF</td>
<td>0.9978</td>
<td>1.0000</td>
<td>0.9512</td>
<td>0.9750</td>
<td>0.0000</td>
<td>0.0060</td>
</tr>
<tr>
<td>Goodware vs. Rooting</td>
<td>LF(CG, SD, API)</td>
<td>21</td>
<td>-</td>
<td>RF</td>
<td>0.9997</td>
<td>1.0000</td>
<td>0.9512</td>
<td>0.9750</td>
<td>0.0000</td>
<td>0.0060</td>
</tr>
<tr>
<td>Goodware vs. Banking Trojan</td>
<td>LF(CG, SD, API)</td>
<td>30</td>
<td>-</td>
<td>RF</td>
<td>0.9987</td>
<td>0.9904</td>
<td>0.9810</td>
<td>0.9856</td>
<td>0.0096</td>
<td>0.0066</td>
</tr>
<tr>
<td>Goodware vs. Spyware</td>
<td>LF(CG, SD, API)</td>
<td>30</td>
<td>-</td>
<td>RF</td>
<td>0.9982</td>
<td>0.9910</td>
<td>0.9831</td>
<td>0.9870</td>
<td>0.0090</td>
<td>0.0181</td>
</tr>
<tr>
<td>Other-malware vs. Rooting</td>
<td>LF(CG, SD, API)</td>
<td>27</td>
<td>-</td>
<td>RF</td>
<td>0.9849</td>
<td>0.9221</td>
<td>0.9595</td>
<td>0.9403</td>
<td>0.0779</td>
<td>0.0457</td>
</tr>
<tr>
<td>Other-malware vs. Banking Trojan</td>
<td>LF(CG, SD, API)</td>
<td>21</td>
<td>-</td>
<td>RF</td>
<td>0.9967</td>
<td>0.9115</td>
<td>0.9537</td>
<td>0.9321</td>
<td>0.0885</td>
<td>0.0510</td>
</tr>
<tr>
<td>Other-malware vs. Spyware</td>
<td>LF(CG, SD, API)</td>
<td>30</td>
<td>-</td>
<td>RF</td>
<td>0.9978</td>
<td>0.9825</td>
<td>0.9867</td>
<td>0.9846</td>
<td>0.0175</td>
<td>0.0145</td>
</tr>
<tr>
<td>SET 4</td>
<td>LM for LM, FC, CG for FC and CG. When more than one of LM, FC, and CG are used at the same time, their Ns are set to the same value. “Distance” column is used to state the best distance measure for classifiers with LM features, and “Classifier” stands for the classifier selected with the best performance.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goodware vs. Rooting</td>
<td>LF(LM, FC, CG, SD, API)</td>
<td>30</td>
<td>Hamming</td>
<td>RF</td>
<td>0.9950</td>
<td>1.0000</td>
<td>0.9750</td>
<td>0.9873</td>
<td>0.0000</td>
<td>0.0033</td>
</tr>
<tr>
<td>Goodware vs. Banking Trojan</td>
<td>LF(LM, FC, CG, SD, API)</td>
<td>27</td>
<td>Euclidean</td>
<td>RF</td>
<td>0.9992</td>
<td>1.0000</td>
<td>0.9905</td>
<td>0.9952</td>
<td>0.0000</td>
<td>0.0033</td>
</tr>
<tr>
<td>Goodware vs. Spyware</td>
<td>LF(LM, FC, CG, SD, API)</td>
<td>21</td>
<td>Hamming</td>
<td>RF</td>
<td>0.9991</td>
<td>1.0000</td>
<td>0.9906</td>
<td>0.9953</td>
<td>0.0000</td>
<td>0.0098</td>
</tr>
<tr>
<td>Other-malware vs. Rooting</td>
<td>LF(LM, FC, CG, SD, API)</td>
<td>21</td>
<td>Hamming</td>
<td>RF</td>
<td>0.9947</td>
<td>0.9545</td>
<td>0.9333</td>
<td>0.9438</td>
<td>0.0454</td>
<td>0.0147</td>
</tr>
<tr>
<td>Other-malware vs. Banking Trojan</td>
<td>LF(LM, FC, CG, SD, API)</td>
<td>24</td>
<td>Euclidean</td>
<td>RF</td>
<td>0.9689</td>
<td>0.9487</td>
<td>0.9569</td>
<td>0.9528</td>
<td>0.0513</td>
<td>0.0532</td>
</tr>
<tr>
<td>Other-malware vs. Spyware</td>
<td>LF(LM, FC, CG, SD, API)</td>
<td>27</td>
<td>Hamming</td>
<td>RF</td>
<td>0.9981</td>
<td>0.9840</td>
<td>0.9885</td>
<td>0.9862</td>
<td>0.0106</td>
<td>0.0125</td>
</tr>
</tbody>
</table>

below) in the no attack case. Each of the 5 settings is defined below. LM for LM, FC, CG for FC and CG. When more than one of LM, FC, and CG are used at the same time, their Ns are set to the same value. “Distance” column is used to state the best distance measure for classifiers with LM features, and “Classifier” stands for the classifier selected with the best performance.

**SET 1 Baseline: Basic Features Only.** Due to a large number of SD features, we first compared the performance of classifiers with all or part of SD features using feature selection methods. We found that a certain number of selected features yielded the best F1 score. This is done via a standard ablation test. In ablation testing, we

---

7 We use “SD” to refer to static and dynamic features, “API” to refer to API package call features, “LM” to refer to the landmark-based features (furthermore, we use “-Rand”, “-Cluster” and “-Max-dis” to represent the three types of landmark selection methods), “FC” corresponds to feature value clustering-based features, and “CG” is for correlation graph-based features. We use “LF(···,···)” denotes the late fusion classifier with appropriate feature inputs. Of the various metrics reported, the most important one is the “F1-score”, which reflects a balance of precision and recall. The column N stands for the number of landmarks or clusters used by LM, FC, and CG.
first compute the performance (F1-score) with all features; we then drop 1 feature and see which feature leads to the biggest drop in performance — this feature, $f_1$ is the most important. We then repeat this process to find the second most important feature $f_2$ (which is the feature that leads to the biggest drop in performance, assuming $f_1$ is already dropped), the third most important feature $f_3$, and so forth. For each $f_j$, we compute the performance of the classifiers using the features in $F_j = F - \{f_1, \ldots, f_j\}$. For each $F_j$, we compute the performance of our classifiers using just the features in $F - F_j$, and choose the $j$ that leads to the highest performance. When distinguishing between rooting malware and goodware, we found that $j = 50$ selected features lead to the best F1 score 0.9195. We then trained classifiers with API features only, as well as the combination of SD and API features (row “SD + API”) respectively. The results of combining SD and API features (better than using SD or API features alone) in Tables 5.8 and 5.9 (SET1) show that the baselines achieve F1-scores of 88.87-96.97% and 91.96-98.26% on the No-Isomorphic and Isomorphic datasets respectively. These are the numbers that FARM has to beat.

**SET 2 FARM w/ LM: FARM with Landmark based Features.** Our SET 2 experiments first compared FARM with LM-features alone while changing the landmark selection method and varying the number of landmarks $LM$. Of the three landmark selection methods, we found that the max-distance heuristic selection (LM-Max_dis) is both not competitive and far more time-consuming. We, therefore, abandoned this method in the following experiments. Next, we compared the remaining two landmark selection methods by combining them with SD, API, and SD + API features respectively. The results show that FARM with landmark features alone beats the baseline in all 12 cases with F1-Scores of 93.78-98.26% and 94.06-99.08% for the No-Isomorphic and Isomorphic datasets respectively.
**SET 3 FARM w/ FC: FARM with Feature Value Clustering based Features.**
The SET 3 experiments used FARM with classifiers trained on data generated using the feature clustering-based features (w.r.t. different number of clusters \( G \) and the one with the best performance is presented in \( N \) column) and combine them with SD, API, and SD + API features respectively. Our SET 3 results show that FARM obtains F1-scores of 92.08-98.6% and 94.22-99.06% respectively. Here again, FARM beats the baseline in all 12 experiments and returns results comparable to those generated by LM-features.

**SET 4 FARM w/ CG: FARM with Correlation Graph based Features.** In SET 4 experiments, we trained our classifiers with the correlation graph-based features (w.r.t. different number of clusters \( G \)) and combined them with SD, API, and SD + API features respectively. Our results show that FARM with CG features beats the baselines on all 12 cases and achieves F1-scores of 94.03-98.56% and 92.96-99.55% on the No-Isomorphic and Isomorphic datasets respectively.

**SET 5 FARM w/ all: FARM Approach with All Transformed Features.**
In SET 5 experiments, we used features from all the proposed feature transformation methods and combined them with SD, API, and SD + API features respectively. The experimental results show that FARM achieves F1-scores of 94.38-99.53% and 95.61-99.69%, again beating out the baselines on all 12 problems. Moreover, the combination of all three feature transformations generated the best results in all.

*Statistical Significance.* We tested the null hypothesis that the best baseline for each of the 12 problems considered was generated by the same underlying process as the best setting of FARM (i.e. with \( LF(LM, FC, CG, SD, API) \).) The null hypothesis was rejected in all 12 cases with \( p \leq 3.5337e-3 \) in all cases, i.e. the probability that the same underlying process generated both the best baseline results and the best FARM results is so low that it is almost zero. Thus, the claim that FARM is better than the
best baseline in distinguishing across the 12 problems considered is statistically valid.

Table 5.10: Statistical Results p-value of best settings of FARM over the best baseline

<table>
<thead>
<tr>
<th>Classification Problem</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goodware vs. Rooting (No-Isomorphic)</td>
<td>1.0959e−10</td>
</tr>
<tr>
<td>Goodware vs. Rooting (Isomorphic)</td>
<td>1.0539e−6</td>
</tr>
<tr>
<td>Goodware vs. Banking Trojan (No-Isomorphic)</td>
<td>2.5891e−7</td>
</tr>
<tr>
<td>Goodware vs. Banking Trojan (Isomorphic)</td>
<td>1.6523e−4</td>
</tr>
<tr>
<td>Goodware vs. Spyware (No-Isomorphic)</td>
<td>8.1829e−6</td>
</tr>
<tr>
<td>Goodware vs. Spyware (Isomorphic)</td>
<td>4.4940e−4</td>
</tr>
<tr>
<td>Other-malware vs. Rooting (No-Isomorphic)</td>
<td>3.5337e−3</td>
</tr>
<tr>
<td>Other-malware vs. Rooting (Isomorphic)</td>
<td>3.3029e−7</td>
</tr>
<tr>
<td>Other-malware vs. Banking Trojan (No-Isomorphic)</td>
<td>7.6698e−5</td>
</tr>
<tr>
<td>Other-malware vs. Banking Trojan (Isomorphic)</td>
<td>1.4564e−6</td>
</tr>
<tr>
<td>Other-malware vs. Spyware (No-Isomorphic)</td>
<td>2.7297e−4</td>
</tr>
<tr>
<td>Other-malware vs. Spyware (Isomorphic)</td>
<td>3.6931e−4</td>
</tr>
</tbody>
</table>

5.3.4. Robustness Evaluation

The goal of the three feature transformations introduced in this paper is to make FARM more robust in the presence of adversarial attacks. We can be sure that malicious hackers will adapt their malware once they realize that it has been detected and that anti-virus engines have developed signatures to protect Android devices from the threat. Though it is impossible to imagine all the types of evasion methods that malicious hackers might come up with, we tested the robustness of FARM against three kinds of attacks.

Threat Model. We assume that the adversary: (i) knows all the 1058 basic features used by FARM, and (ii) that the adversary is also familiar with the suite of 8 classifiers used in the paper (Bernoulli and Gaussian Naive Bayes, Random Forest, k-Nearest Neighbor, Logistic Regression, Adaboost, Gradient Boosted Decision Tree, XGB, and SVM). We further assume that the attacker has read this paper and hence knows about the three types of feature transformation used. But we do not assume the attacker knows any of the following: (i) the specific landmarks used, the landmark selection strategy used and/or distance function used by the defender in the Landmark-based
Feature transformation, (ii) the number of clusters and the $\oplus$ feature combination algorithm used in the Feature-Value-based Clustering Transformation, and (iii) the number of groups and the specific $\oplus$ operator used by the defender in the Correlation-Graph based feature transformation. We further assume that the attacker carries out the three kinds of attacks described below. \footnote{We assume the attacker tries three kinds of attacks:}

(a) Fake API Package calls in which the adversary injects irrelevant API package calls into his malware.

(b) Fake permission requests in which the adversary requests permissions that are irrelevant for his malware.

(c) Reduced API Package calls in which the adversary tries to artificially reduce the number of calls made to API packages.

\textit{Note that it is more or less impossible to imagine all the types of attacks that a savvy attacker may come up with - hence, in this paper, we limit our claims of robustness to these types of attacks.}

\textit{Fake API package call feature attack.} Here, attackers try to evade FARM by increasing the percentage of fake API package calls made, i.e. by adding more and more fake API package calls into the code. Table 5.11 shows that the impact of this attack on FARM is just 10.47-72.12% than the impact on the baselines — on average, across the 12 classification problems, the impact on FARM is 36%, i.e. FARM is about 3 times as robust as the baselines across the 12 problems studied in this paper. Figures 5.10 and 5.11 show the impact of this attack on the best version of FARM (blue line with square markers) compared to the best baseline (yellow line with

\footnote{We do not claim that FARM is robust against all kinds of adversarial attacks (e.g. obfuscated gradient attacks \cite{obfuscated-gradient-attacks}). Indeed, such a claim would be very hard to justify for almost any paper without making some unrealistic assumptions.}
Table 5.11: Average impact score $a$ of FARM over the best baseline on increased fake API package call attack

<table>
<thead>
<tr>
<th>Classification Problem</th>
<th>$a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goodware vs. Rooting (No-Isomorphism)</td>
<td>0.1846</td>
</tr>
<tr>
<td>Goodware vs. Rooting (Isomorphic)</td>
<td>0.6824</td>
</tr>
<tr>
<td>Goodware vs. Banking Trojan (No-Isomorphism)</td>
<td>0.1047</td>
</tr>
<tr>
<td>Goodware vs. Banking Trojan (Isomorphic)</td>
<td>0.7212</td>
</tr>
<tr>
<td>Goodware vs. Spyware (No-Isomorphism)</td>
<td>0.1138</td>
</tr>
<tr>
<td>Goodware vs. Spyware (Isomorphic)</td>
<td>0.4505</td>
</tr>
<tr>
<td>Other-malware vs. Rooting (No-Isomorphic)</td>
<td>0.2310</td>
</tr>
<tr>
<td>Other-malware vs. Rooting (Isomorphic)</td>
<td>0.2989</td>
</tr>
<tr>
<td>Other-malware vs. Banking Trojan (No-Isomorphism)</td>
<td>0.1728</td>
</tr>
<tr>
<td>Other-malware vs. Banking Trojan (Isomorphic)</td>
<td>0.5348</td>
</tr>
<tr>
<td>Other-malware vs. Spyware (No-Isomorphism)</td>
<td>0.1047</td>
</tr>
<tr>
<td>Other-malware vs. Spyware (Isomorphic)</td>
<td>0.7212</td>
</tr>
</tbody>
</table>

dot markers) as the percentage of fake calls increases in the rooting app vs. goodware and rooting app vs. other malware classification problems respectively.

Surprisingly, as more fake API package calls are made, it becomes easier for classifiers to identify rooting malware. This suggests that the malicious behavior of malware is related to the API package calls that they make. For example, Android Banking Trojans call API `android.app.admin` to hijack a smartphone’s administrative features at the system level, while it is not commonly called in Android Goodware. Thus, when we simulate the attacker’s behavior and increasing the Fake Call Percentage in malware, the performance of the classifier using both our best setting and baseline improves because the fake calls may involve the malware calls many more API calls than a piece of goodware would ordinarily make. FARM always achieves better F1 performance, especially when the attacker injects only a small percentage of fake API package calls (which is the best strategy for him as this is when both FARM and the baselines’ predictive accuracy is lowest in this situation).

Fake permission attack. Second, we assume that attackers try to evade malware detection by increasing the number of permissions they seek. Table 5.12 shows that on average, the impact of this attack on FARM is 10.25-74.42% of the impact on the
Table 5.12: Average impact score $a$ of FARM over the best baseline on increased percentage of permissions attack

<table>
<thead>
<tr>
<th>Classification Problem</th>
<th>$a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goodware vs. Rooting (No-Isomorphic)</td>
<td>0.1952</td>
</tr>
<tr>
<td>Goodware vs. Rooting (Isomorphic)</td>
<td>0.6063</td>
</tr>
<tr>
<td>Goodware vs. Banking Trojan (No-Isomorphic)</td>
<td>0.1025</td>
</tr>
<tr>
<td>Goodware vs. Banking Trojan (Isomorphic)</td>
<td>0.7442</td>
</tr>
<tr>
<td>Goodware vs. Spyware (No-Isomorphic)</td>
<td>0.1376</td>
</tr>
<tr>
<td>Goodware vs. Spyware (Isomorphic)</td>
<td>0.4623</td>
</tr>
<tr>
<td>Other-malware vs. Rooting (No-Isomorphic)</td>
<td>0.2286</td>
</tr>
<tr>
<td>Other-malware vs. Rooting (Isomorphic)</td>
<td>0.2617</td>
</tr>
<tr>
<td>Other-malware vs. Banking Trojan (No-Isomorphic)</td>
<td>0.1791</td>
</tr>
<tr>
<td>Other-malware vs. Banking Trojan (Isomorphic)</td>
<td>0.5250</td>
</tr>
<tr>
<td>Other-malware vs. Spyware (No-Isomorphic)</td>
<td>0.1025</td>
</tr>
<tr>
<td>Other-malware vs. Spyware (Isomorphic)</td>
<td>0.7442</td>
</tr>
</tbody>
</table>

best baseline, with the average impact on FARM being 35.74%. Thus, as in the case of the first attack, FARM is about 3 times as robust to this attack than the best baseline.

The “fake permission” percentage in Figures 5.12 and 5.13 refer to the percentage of requested permissions that are fake. The figures respectively show the results of distinguishing between rooting malware and goodware on the one hand, and other malware on the other hand. We see that as more permissions are required, both FARM and the baselines do a better job in detecting rooting malware. But again, the best-case scenario for the attacker is when the percentage of fake (unused) permissions is below about 8%, and in this case, FARM beats the baselines. FARM performs better than the best baseline because malware also achieves its malicious function by calling system permissions in the manifest file. For example, Android Spyware uses system permissions permission:RECEIVE_SMS and permission:READ_SMS to steal messages from the smartphone while common Android Goodware does not. Also, both common Android Goodware and Android Spyware do not call the permission android.permission.SET_TIME, which allows the application to set the system time.

When the attacker behavior increases the Unused Permissions Percentage, it the classifiers’ job becomes easier to distinguish the adapted malware if it calls the permission
Detection and Analysis of Android Malware Families

`android.permission.SET_TIME`. The performance of both classifiers increases at the same time.

Table 5.13: Average impact score $a$ of FARM over the best baseline on reduced percentage of API package call attack

<table>
<thead>
<tr>
<th>Classification Problem</th>
<th>$a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goodware vs. Rooting (No-Isomorphic)</td>
<td>-0.7319</td>
</tr>
<tr>
<td>Goodware vs. Rooting (Isomorphic)</td>
<td>-3.2924</td>
</tr>
<tr>
<td>Goodware vs. Banking Trojan (No-Isomorphic)</td>
<td>-1.1666</td>
</tr>
<tr>
<td>Goodware vs. Banking Trojan (Isomorphic)</td>
<td>0.6750</td>
</tr>
<tr>
<td>Goodware vs. Spyware (No-Isomorphic)</td>
<td>-3.6451</td>
</tr>
<tr>
<td>Goodware vs. Spyware (Isomorphic)</td>
<td>-0.9673</td>
</tr>
<tr>
<td>Other-malware vs. Rooting (No-Isomorphic)</td>
<td>0.2623</td>
</tr>
<tr>
<td>Other-malware vs. Rooting (Isomorphic)</td>
<td>6.1428</td>
</tr>
<tr>
<td>Other-malware vs. Banking Trojan (No-Isomorphic)</td>
<td>-0.0247</td>
</tr>
<tr>
<td>Other-malware vs. Banking Trojan (Isomorphic)</td>
<td>-0.0212</td>
</tr>
<tr>
<td>Other-malware vs. Spyware (No-Isomorphic)</td>
<td>-1.1666</td>
</tr>
<tr>
<td>Other-malware vs. Spyware (Isomorphic)</td>
<td>0.6750</td>
</tr>
</tbody>
</table>

Reduction API feature attack. Third, we assume that attackers are more strategic and capable — we allow them to selectively drop some API package calls by 1 when the original value is at least 2. Table 5.13 shows that the impact of this attack on FARM ranges from -3.64-6.14, suggesting a wide variation. On 11 of 12 cases, FARM outperforms the best baseline, but in one case (Other malware vs. Rooting malware), the best baseline outperforms FARM. Again, on average, FARM performs very well, with the accuracy of FARM often improving under this attack. This is because the modified malware achieves its malicious purpose by calling specific API calls and because the number of called APIs can be decreased but cannot be fully removed.

The results on Rooting malware detection are shown in Figures 5.14 and 5.15. Unlike the results from the previous two kinds of attack, we see that the reduced API feature attack is harder for both FARM and traditional classifiers to adapt to. However, the situation is worse for the baseline classifiers. When distinguishing between rooting malware and goodware, the F1 performance goes down slightly as the number of API package calls is reduced. The reason might be that rooting malware gets less
malicious and more similar to goodware in this case. However, the F1 score goes up when distinguishing rooting malware from other malware. The reason might be that as rooting malware is getting more similar to goodware, it ends up being more distinct from other malware. Table 5.11, Table 5.12 and Table 5.13 show 3 kinds of applied attack during the robustness test, an impact score is calculated according to the average of 0% to 20% increased or decreased number of APIs or permissions. The performance under attack is always increasing because our classifiers distinguish Goodware vs. Malware on API or Permission features. When we simulate the attack, we increase the number of unused APIs or permissions in malware, and so can detect the malware easier because some API or permission features may not be used by both goodware or malware, but now more malware calls the common unused feature, leading to better classification results. Again, FARM performs better than the baseline. When we decrease 1 for some of called APIs (frequency ≥ 2 to keep its malicious function) in malware, no obvious change in the performance because the feature space doesn’t change too much compared to the previous two attacks. At the same time, FARM still has better performance.
Figure 5.10: Impact of fake API package call attack on Android rooting malware detection: Goodware vs. Rooting Malware (No-Isomorphic)

Figure 5.11: Impact of fake API package call attack on Android rooting malware detection: Other Malware vs. Rooting Malware (No-Isomorphic)

Figure 5.12: Impact of increased percentage of permissions attack on Android rooting malware detection: Goodware vs. Rooting Malware (No-Isomorphic)

Figure 5.13: Impact of increased percentage of permissions attack on Android rooting malware detection: Other Malware vs. Rooting Malware (No-Isomorphic)

Figure 5.14: Impact of reduced percentage of API package call attack on Android rooting malware detection: Goodware vs. Rooting Malware (No-Isomorphic)

Figure 5.15: Impact of reduced percentage of API package call attack on Android rooting malware detection: Other Malware vs. Rooting Malware (No-Isomorphic)
Detection and Analysis of Android Malware Families

Figure 5.16: Impact of fake API package call attack on Android Banking Trojans detection: Goodware vs. Banking Trojans (Isomorphic)

Figure 5.17: Impact of fake API package call attack on Android Banking Trojans detection: Other Malware vs. Banking Trojans (Isomorphic)

Figure 5.18: Impact of increased percentage of permissions attack on Android Banking Trojans detection: Goodware vs. Banking Trojans (Isomorphic)

Figure 5.19: Impact of increased percentage of permissions attack on Android Banking Trojans detection: Other Malware vs. Banking Trojans (Isomorphic)

Figure 5.20: Impact of reduced percentage of API package call attack on Android Banking Trojans detection: Goodware vs. Banking Trojans (Isomorphic)

Figure 5.21: Impact of reduced percentage of API package call attack on Android Banking Trojans detection: Other Malware vs. Banking Trojans (Isomorphic)
Figure 5.22: Impact of fake API package call attack on Android Spyware detection: Goodware vs. Spyware (No-Isomorphic)

Figure 5.23: Impact of fake API package call attack on Android Spyware detection: Other Malware vs. Spyware (No-Isomorphic)

Figure 5.24: Impact of increased percentage of permissions attack on Android Spyware detection: Goodware vs. Spyware (No-Isomorphic)

Figure 5.25: Impact of increased percentage of permissions attack on Android Spyware detection: Other Malware vs. Spyware (No-Isomorphic)

Figure 5.26: Impact of reduced percentage of API package call attack on Android Spyware detection: Goodware vs. Spyware (No-Isomorphic)

Figure 5.27: Impact of reduced percentage of API package call attack on Android Spyware detection: Other Malware vs. Spyware (No-Isomorphic)
Chapter 6

SAAM: Stability Analysis of Android Malware Families

Section 6.1

Introduction

Android is one of the most popular operating systems nowadays. Android devices provide high-performance services for human users while also storing sensitive and personal information. As a result, Android malware has been actively used by cybercriminals to execute malicious behavior on Android platforms.

Based on the nature of the malicious behavior, Android malware samples are categorized into different types (such as banking trojans, spyware, and rooting malware). Multiple malware families persist in each malware category. For instance, FakeBank, AceCard, Svpeng are all well known families of Android Banking trojans. Likewise, SMSspy and SpyOO are well known families of Android spyware. A collection of Android malware from the same family is produced from the same source code base. A malware family can evade the constantly improving malware detection techniques and persist for several years by continuously evolving. Studying the underlying malware
family evolvement is essential for defending the Android system.

There are some existing efforts from academia and industry on malware families [92, 59, 93, 94]. However, they focus on summarizing the patterns or behaviors of malware samples in the same family. As far as we know, there is no existing research on how malware samples from the same family evolve to cope with the environment change and the innovation of malware detection techniques. We aim to fix this gap and execute research on 122 popular Android malware families and 120 popular Android goodware families.

Intuitively, the features of malware samples from the same family would show different characteristics according to malware’s function and target.

(a) Some features are mainly related to the APK’s function, e.g., to play music. Since malware samples from the same family might disguise themselves with different functionality, these features are hard to predict.

(b) Some features are closely related to the APK’s target, i.e., the malware family’s malicious behavior. These features might be (1) relatively stable over time to achieve the malware’s purpose; (2) evolving or diminishing over time to cope with emerging detection techniques, especially after the family is named.

We extract features for Android applications and define stability for features based on its change on a set of variants from the same malware/goodware family. By analyzing the properties of different features, we observe some interesting phenomenons presented in this work.

Section 6.2

Feature Vector Partition

Assume that there are \( n \) applications from the same family sorted by the first-seen-date \([t_1, ..., t_n]\) \((t_i \leq t_{i+1})\). We extract some features for each application and study
SAAM: Stability Analysis of Android Malware Families

Figure 6.1: Example signal

how these features evolve over time. For one feature, we denote its value for the $n$ applications as $[x_1, \ldots, x_n]$, where $x_i$ is the $i^{th}$ application’s feature value. We treat the time-series value $[x_1, \ldots, x_n]$ as a signal $X$.

**Example 1.** For better explanation and illustration, we select 11 APKs from a malware family named Boogr and use their “android.telephony” feature values to generate an example signal. Figure 6.1 plots the signal with respect to the first-seen date of APKs.

**Definition 1** (Rate of Change). The rate of change (ROC) of a signal $X$ from time $t_i$ to time $t_j$ is

$$ROC(X, i, j) = \frac{x_j - x_i}{t_j - t_i}$$

The signal in Example 1 is $X = [x_1, \ldots, x_{11}]$. Specifically, $x_1 = 11, x_3 = 16$ and $t_3 - t_1 = 147$ with “day” as the unit. Then we can get $ROC(X, 1, 3) = (x_3 - x_1)/(t_3 - t_1) = 0.034$.

We would define a signal’s stability score according to its partition and the ROC on each part of the partition. First, we define *partition* as follows.

**Definition 2** (Partition). For a given signal $X = [x_1, \ldots, x_n]$, where $x_i$ are sorted in ascending order according to their timestamp $t_i$, a partition $\mathcal{P}$ is derived by separating
SAAM: Stability Analysis of Android Malware Families

X into a set of disjoint signal segments.

For a given partition, there are two properties.

(a) Any pair of adjacent segments share a common point. Formally, if \([x_{j_a}, \ldots, x_{j_b}]\) and \([x_{j_c}, \ldots, x_{j_d}]\) are two adjacent segments from a partition of \(X\), then \(j_b = j_c\).

(b) The start point of the first segment is \(x_1\) and the end point of the last segment is \(x_n\).

Note that a partition \(\mathcal{P}\) with \(m - 1\) segments can be denoted by \(m\) check points (i.e., the start and end points of all segments), thus we define it as \(\mathcal{P} = [pt_1, \ldots, pt_m] \subset [1, \ldots, n]\), where \(pt_1 = 1, pt_m = n\). Apparently, \(|\mathcal{P}| \geq 2\) because it always includes 1 and \(n\).

Recall Example 1, a partition \(\mathcal{P} = [1, 4, 11]\) means \(X\) is divided into 2 segments, \([x_1, \ldots, x_4]\) and \([x_4, \ldots, x_{11}]\).

**Definition 3** (\(\tau\)-Homogeneous Partition). A \(\tau\)-homogeneous partition of a signal satisfies following constraints.

(a) **Monotonicity.** Within each segment \([x_{pt_{j-1}}, \ldots, x_{pt_j}]\), there should be \(x_{pt_{j-1}} \leq \ldots \leq x_{pt_j}\) or \(x_{pt_{j-1}} \geq \ldots \geq x_{pt_j}\).

(b) **Homogeneity.** The absolute ROC value of all segments should be bounded within a narrow interval, i.e., \(|ROC(X, pt_{j-1}, pt_j)| \leq \tau\).

Recall Example 1, the partition \(\mathcal{P} = [1, 4, 11]\) is not \(\tau\)-homogeneous because the second segment \([x_4, \ldots, x_{11}]\) is not monotonic. Instead, \(\mathcal{P} = [1, 4, 8, 10, 11]\) is \(\tau\)-homogeneous.

Note that for a signal \(X\) and a \(\tau\)-homogeneous partition \(\mathcal{P}\), a new partition \(\mathcal{P}' = \mathcal{P} \cup v, \forall v \in \mathcal{N}\setminus \mathcal{P}\) may be **not** a \(\tau\)-homogeneous partition.

Suppose \(pt_{j-1} \leq v \leq pt_j\), then \(\mathcal{P}'\) leads to \(m\) segments, \(m - 2\) of which are exactly the same as that of partition \(\mathcal{P}\) and they obey Definition 3. However, the other two
segments \([x_{pt_j-1}, \ldots, x_v]\) and \([x_v, \ldots, x_{pt_j}]\) may not obey homogeneity constraint of the definition. We can use a simple counter-example to show that. If \(t_v - t_{pt_j-1} \to 0\) and \(x_v - x_{pt_j-1} > 0\), then \(\text{ROC}(X, pt_j-1, v) \to \infty\). Thus, there is no way to guarantee \(\text{ROC}(X, pt_j-1, v) < \tau\).

**Proposition 1** For a signal \(X\) and a \(\tau\)-homogeneous partition \(P = [pt_1, \ldots, pt_m]\), if there are two adjacent segments \([x_{pt_j-1}, \ldots, x_{pt_j}]\) and \([x_{pt_j}, \ldots, x_{pt_{j+1}}]\) with the same monotonicity direction (increasing or decreasing), we can get a new \(\tau\)-homogeneous partition \(P'' = P \setminus \{pt_j\}\).

**Proof.** The new partition \(P''\) has \(m - 1\) check points and thus \(m - 2\) segments, \(m - 3\) of which are the same with that of the original partition. Thus we only need to prove the new segment \([x_{pt_j-1}, \ldots, x_{pt_{j+1}}]\) obeys Definition 3.

(a) **Monotonicity.** Since segments \([x_{pt_j-1}, \ldots, x_{pt_j}]\) and \([x_{pt_j}, \ldots, x_{pt_{j+1}}]\) has the same monotonicity direction (increasing or decreasing), we can easily get \([x_{pt_{j}}, \ldots, x_{pt_{j+1}}]\) are monotonic.

(b) **Homogeneity.** Assume that we denote \(\frac{x_{pt_j} - x_{pt_j-1}}{t_{pt_j} - t_{pt_j-1}} = \frac{\Delta x_1}{\Delta t_1}\) and \(\frac{x_{pt_{j+1}} - x_{pt_j}}{t_{pt_{j+1}} - t_{pt_j}} = \frac{\Delta x_2}{\Delta t_2}\).

From the properties of partition \(P\), both \(|\frac{\Delta x_1}{\Delta t_1}|\) and \(|\frac{\Delta x_2}{\Delta t_2}|\) are less or equal than \(\tau\). The rate of change of the segment \([x_{pt_j-1}, \ldots, x_{pt_{j+1}}]\) from \(P''\) is

\[
\text{ROC}(X, pt_{j-1}, pt_{j+1}) = \frac{x_{pt_{j+1}} - x_{pt_j}}{t_{pt_{j+1}} - t_{pt_j}} = \frac{\Delta x_1 + \Delta x_2}{\Delta t_1 + \Delta t_2}.
\]

Assume that \(|\frac{\Delta x_1}{\Delta t_1}| \leq |\frac{\Delta x_2}{\Delta t_2}| \leq \tau\), we can get \(|\Delta x_1| \leq \Delta t_1 |\frac{\Delta x_2}{\Delta t_2}|\), thus

\[
|\text{ROC}(X, pt_{j-1}, pt_{j+1})| \leq \frac{\Delta t_1 |\frac{\Delta x_2}{\Delta t_2}| + |\Delta x_2|}{\Delta t_1 + \Delta t_2} = \frac{|\Delta x_2|}{\Delta t_2} \leq \tau.
\]

Similarly, we can show \(|\text{ROC}(X, pt_{j-1}, pt_{j+1})| \leq \tau\) also holds when \(\tau \geq |\frac{\Delta x_1}{\Delta t_1}| \geq |\frac{\Delta x_2}{\Delta t_2}|\). Therefore, the new segment of \(P''\) satisfies the homogeneity constraint. \(\Box\)
We want to find a homogeneous partition that satisfies two requirements: (i) keeps \( \tau \) as small as possible so that the change within a segment is small, and (ii) keeps the number of segments small so that changes across segments is small. We therefore associate a score \( \alpha \tau + (1 - \alpha)\frac{|P|}{|X|} \). We can then introduce a formal definition of an “\((\alpha, \tau)\)-optimal partition” to be the one that minimizes this objective function.

**Definition 4 \(((\alpha, \tau)\)-Optimal Partition\).** The \((\alpha, \tau)\)-optimal partition of a signal is a \(\tau\)-homogeneous partition that has minimum weighted sum of the upper-bound of segment rate of change and the number of segments, i.e., \(\alpha \tau + (1 - \alpha)\frac{|P|}{|X|}\) with \(\tau \in [0, 1]\), where \(\alpha, 1 - \alpha\) are the corresponding weights.

The \((\alpha, \tau)\)-optimal partition is computed via the following optimization problem \(\text{OHP}\).

\[
\text{OHP} : \min_{P, \tau} \alpha \tau + (1 - \alpha)\frac{|P|}{|X|} \quad (6.1)
\]

s.t.
\[
M_t(X, pt_{j-1}, pt_j) = 1, \forall j \in \{1, ..., m + 1\} \quad (6.2)
\]
\[
|\text{ROC}(X, pt_{j-1}, pt_j)| \leq \tau, \forall j \in \{1, ..., m + 1\} \quad (6.3)
\]

Note that \(M_t(X, pt_{j-1}, pt_j)\) is the function to test the monotonicity of the \(j^{th}\) segment. \(M_t(X, pt_{j-1}, pt_j) = 1\) if and only if values in \([x_{pt_{j-1}}, x_{pt_j}]\) monotonically increase or decrease.

**Theorem 1.** The solution of \(\text{OHP}\) is the \((\alpha, \tau)\)-optimal partition with the minimum \(|P|\) for any given \(\alpha \in [0, 1]\).

**Proof.** For a given signal \(X\), let us use \(\text{obj}(P, \tau)\) to denote the objective value of the \(\text{OHP}\) with given variables \(P\) and \(\tau\). Obviously, the partition \(P = X \setminus \{x_1, x_n\}\), which leads to \(n - 1\) segments is the largest partition and \(\tau\)-homogeneous. From Proposition 6.2 we can see that if there are adjacent segments of the same monotonicity direction,
combining a pair of them will never increase the $\tau$ value while the number of segments will decrease by 1. Thus, if we use $pt_j$ to denote the check point that separates the pair of segments, obviously $\text{obj}(P\{pt_j\}, \tau) < \text{obj}(P, \tau)$. Therefore, we can iterate this process until all adjacent segments with identical monotonicity are combined, during which the partition size $|P|$ is minimized. Meanwhile, the objective value is minimized.

Therefore, we can find the solution of OHP using Algorithm 5.

**Algorithm 5 Algorithm for OHP**

```
Input: X
Output: P
Initialize: P = ∅
if |X| < 3 then
    return P
// Set the left and right indices of the current partition
ind_l = 1, ind_r = 2
while ind_r < n do
    if $(x_{ind_r+1} - x_{ind_r})(x_{ind_r} - x_{ind_l}) > 0$ then
        ind_r += 1
    else
        ind_l = ind_r, ind_r = ind_r + 1
    P = P ∪ {ind_l}
Return: P
```

Then, we define stability score $ss$ based on the optimal homogeneous partition of a signal.

**Definition 5 (Stability Score).** Given a signal $X$ and its $\tau$-homogeneous partition $P = [pt_1, ..., pt_m]$, we can get $m + 1$ segments from $X$. For the $j^{th}$ segment $[x_{pt_{j-1}}, ..., x_{pt_j}]$, we can get its rate of change $\text{ROC}(X, pt_{j-1}, pt_j)$. We then define the stability score of signal $X$ under partition $P = [pt_1, ..., pt_m]$ as

$$ss(X, P) = 1 - \alpha \tau - (1 - \alpha)\frac{|P|}{|X|}$$
The intuition behind the stability score is that we want both a small $\alpha \tau$ and a small $(1 - \alpha)|P|/|X|$. So we want the sum of these two terms to be as small as possible. As we want the stability score to be larger for more stable features, we subtract their sum from 1, getting the above formula.

Discussion about the normalization of stability score. After the definition of the stability score, we would like to be sure it makes sense when it is used to compare a number of signals. For example, when we want to compare a number of features for a set of samples from the same Android family, and in this case, users would need to normalize the features into $[0, 1]$ and select the suitable time unit such that minimum $t_j - t_{j-1} = 1$. Then there is $\tau \in (0, 1]$ and for each signal the maximum possible $\tau$ is 1. By selecting a working $\alpha \in (0, 1)$ according to users’ requirement, $ss$ is always bounded within $[0, 1]$ and it is comparable among different signals.

Proposition 2 For a given signal $X$ and its $\tau$-Homogeneous Partition $P$, the stability score $ss(X, P)$ is maximized when $P$ is the optimal solution of problem OHP.

Recall that in Example 1, the $(\alpha, \tau)$-optimal partition is $P = [1, 4, 8, 11]$. We compute the ROC in those 3 segments from the partition and get $ROC(X, 1, 4) = 0.1275, ROC(X, 4, 8) = 0.1870, ROC(X, 8, 11) = 0.0591$. Thus there is $\tau = 0.1870$ and the stability score of this signal is $ss = 1 - 0.1870\alpha - (1 - \alpha)2/11 = 0.8182 - 0052\alpha$.

Section 6.3

Signal Smoothing

We note that the homogeneous partition of a signal is vulnerable to noises in the signal. For example, Figure 6.2 shows $x_2$ in the original signal as shown in Figure 6.1 is influenced by the noise and increase to a value greater than $x_3$. In this case, the $(\alpha, \tau)$-optimal partition is now $P = [1, 2, 4, 8, 11]$, which is different from before. The stability score would also be changed. To minimize the potential influence from noises, we adopt signal processing techniques for further analysis.
In particular, we apply the Moving-Average Signal Smoothing method to the feature signal. By smoothing the signal over time, the influence of noises will be decreased. Moving-average technique is a standard one. Specifically, there are two approaches to achieve it on our feature signal, which has \( n \) sorted values in \( X \) with their timestamps \( t_1 \) to \( t_n \).

The first choice is to smooth the signal with respect to a window size \( T \) of time. Each \( x_i \) with \( t_i \geq T + t_1 \) is smoothed as \( \bar{x}_i = \text{ave}_{t' = t_1 - T}^{t_i} x_{i'} \). In this case, the smoothed signal \( \bar{X} \) is shorter than \( X \) as the values with \( t_i < T + t_1 \) are discarded.

Another choice is to smooth the signal without consideration of the time information. Assume there is a window size \( M \), then \( x_i \) will be smoothed as \( \bar{x}_i = \text{ave}_{i' = i - T}^{i} x_{i'} \), which is the average of \( T + 1 \) values.

Despite the moving-average method, we may also try other techniques. For example, Gaussian filter that gets the weighted average using the discretized Gaussian function. For example, \( \bar{x}_i = \sum_{i' = i - T}^{i} x_{i'} g_{i - i'} \), where \( g_{i - i'} \) is the corresponding weight and \( \sum_{i' = i - T}^{i} g_{i - i'} = 1 \).
Section 6.4

Studies and Analysis of SAAM Dataset

6.4.1. Dataset

We select 122 popular malware families according to the published Android Malware Variants lists [95, 96] for our study. For each family (e.g., FakeBank, Svpeng, etc.), we download a number of malware samples (approximately 60 samples in each family) detected between 2012 and 2019 on VirusTotal. We also collect Android malware families’ detection time from Kaspersky[1] and Symantec[2]. Also, our dataset includes the top 120 popular goodware families (e.g., YouTube, Zoom, etc.) from the Top free Apps list on Google Play Store[3], and we also collect approximately 60 samples for each goodware family, and the different samples have different first release date between 2012 to 2020, and we use their first release date as the timestamp in the later experiment. Therefore, we have 14,489 (7,648 malware + 6,841 goodware) samples in total.

After getting the sample datasets, we extract the related features from the analysis report from VirusTotal, and then we union all the feature vectors together to get our final feature file. After the feature combination, we have 885 features in total, and 208 API package features as defined in Google Android Developer Guides[4], 227 Operation code features[5], 346 Permission features[6] and 104 System Command features[7].

[7] https://ss64.com/bash/
6.4.2. Demo of feature signals in Android Malware Families

After defining the stability score in Definition 5, we implement a stability score demo system to visualize the top stable or unstable feature signals as Figure 6.3 and Figure 6.4 shows, where Figure 6.3 shows the top 3 stable feature signals in the Adware malware family with name Airpush and Figure 6.4 displays the top 3 unstable feature signals in the Android Banking Trojans family named Svpeng. According to F-secure, Airpush is a kind of Android application that pushes advertising content to the device’s notification bar intrusively. Svpeng can steal the victim’s bank login information by displaying a fake login interface to intercept the user’s normal request.

In the demo figure system, the user can choose various parameters to visualize the feature signal differently. The first parameter specifies the way to plot the signal, whether the interval between points is divided by the sample itself or by the gap timestamp between the sample. The second parameter decides which malware family’s feature is displayed on the panel, and all 122 malware families’ feature signals are available. The third parameter decides the \( \alpha \) used to calculate the stability score as defined in Definition 5. The fourth parameter controls the numbers of top feature signals to show. The fifth parameter specifies the smoothing method applied to the feature signal as defined in Section 6.3, while the last parameter decides whether to display the top stable or unstable features.

In Figure 6.3 it shows that the top 3 stable features are API package android, android.annotation and android.app.assist, and all their stability score equals 1 with the malware family detection date (the vertical green dotted line) on March 28, 2014. There are 76 samples in the chosen Airpush family, with the sample’s timestamp ranges from November 1, 2011, to August 2, 2019. The feature signal is also smoothed with the time-based sliding window method, and the window size equals 60 days. The stability score parameter \( \alpha \) equals 0.25. Similarly, in Figure 6.4 the top 3 unstable

---


102
features in the Svpeng family are the total detection number of antivirus engines on VirusTotal, the permission of BIND


DEVICE


ADMIN, and the filesize of the malware sample. This time, the top unstable feature signals’ stability score is less than 1 with $\alpha$ equals to 0.75 using the event-based smoothing method with window size equals to 2.

Therefore, with the help of the Stability Score demo system, users can easily distinguish the top stable or unstable feature signals of different Android malware families in our datasets.

Figure 6.3: Demo GUI of Airpush Top stable feature signals

6.4.3. Evaluation Metrics

In this section, we use 3 metrics in result discussion.

- $ave_m(f), ave_g(f)$, the average stability score of the feature $f$ across all malware/goodware families;

- $std_m(f), std_g(f)$, the standard deviation of feature $f$’s stability score across all malware/goodware families;
etp_m(f), etp_g(f), the entropy of feature f’s stability score values across all malware/goodware families. Specifically, since the stability score is a value between 0 and 1, we first discretize the ss values into bins, then compute the entropy.

**Definition 6 (MG_μ ratio).** The MG_μ ratio is a metric to evaluate a feature’s stability among malware versus goodware based on a parameter μ. Here we consider 3 different values of μ, i.e., the stability score’s average ave, standard deviation std and entropy etp. Formally,

\[ MG_\mu(f) = \frac{\mu_m(f)}{\mu_g(f)} \]

Note that MG_ave(f) > 1 (resp. < 1) means the feature is more stable among malware than goodware (resp. the inverse); in contrast, when μ = std or etp, MG_μ(f) has the opposite meaning.
6.4.4. Distribution of Stability Score in Android application families

After calculating the stability score for each feature in each Android application family (122 malware families + 120 goodware families), we plot the distribution of different kinds of features (API Package, Permission, Opcode, and System Command) as the following figures show.

Figure 6.5 shows the distribution of average stability score on API Package features between goodware and malware families. From the figure, we can see that 17.79% of the features have a stability score $\leq 0.85$ in malware families, and no features in goodware families have a stability score $\leq 0.85$. 29.32% of the features have a stability score $\leq 0.9$ in malware families, and in the case of goodware, it is under 15.86%. We can claim that API Package features are more stable in goodware families than malware families according to the definition of stability score with a p-value equals to 0.002, which means the results are significant.

![Figure 6.5: Distribution of average stability score on API Package features](image)

Figure 6.6 shows the distribution of average stability score on Permission features between goodware and malware families. From the figure, we can see that 3.18% of the features have a stability score $\leq 0.9$ in malware families and no features in goodware families have a stability score $\leq 0.9$. 8.96% of the features have a stability score $\leq 0.95$ in malware families, and in the case of goodware, it is under 0.87%. We can claim that Permission features are more stable in goodware families than malware families according to the definition of stability score with a p-value equals to $6.487 \times 10^{-9}$,
which means the results are significant.

Figure 6.6: Distribution of average stability score on Permission features

Figure 6.7 shows the distribution of average stability score on Opcode features between goodware and malware families. From the figure, we can see that 4.41% of the features have a stability score \( \leq 0.75 \) in malware families and no features in goodware families have a stability score \( \leq 0.75 \). 20.26% of the features have a stability score \( \leq 0.77 \) in malware families, and in the case of goodware, it is under 3.08%. We can claim that Opcode features are more stable in goodware families than malware families according to the definition of stability score with a p-value equals to 0.001, which means the results are significant.

Figure 6.7: Distribution of average stability score on Opcode features

Figure 6.8 shows the distribution of average stability score on System Command features between goodware and malware families. From the figure, we can see that 4.81% of the features have a stability score \( \leq 0.9 \) in malware families and 0.96% of features in goodware families have a stability score \( \leq 0.9 \). Both 14.42% of the features
have a stability score \( \leq 0.95 \) in goodware and malware families. This time, the p-value equals 0.592, which is greater than 0.05 when we are comparing the stability scores in the System Command features, so we believe that System Command-based features are equally stable in goodware and malware families according to the definition of stability score.

![Distribution of average stability score on System Command features](image)

**Figure 6.8: Distribution of average stability score on System Command features**

6.4.5. \( MG_\mu \) ratio with average Stability Score, Entropy and Standard Deviation

This section analyzes the \( MG_\mu \) ratio with average stability score, entropy, and standard deviation in our dataset. We want to know if the \( MG_\mu \) ratio can reveal the stability of the general features in the goodware and malware families.

Figure 6.9 shows the \( MG_{avg} \) with average stability score. From the left sub-figure, we can observe that about half features are stable in goodware families, and the other half of the features are relatively stable in malware families. From the right sub-figure, each dot corresponds to one feature, and the diagonal line divides the features with \( MG_{avg} \) greater than or less than 1. Similarly, we can see that near half features have \( MG_{avg} \geq 1 \) and the \( MG_{avg} \) of the other half features \( \leq 1 \). Therefore, we cannot make a strong conclusion on the stability analysis of general features by defining \( MG_{avg} \) here with the p-value equals to 0.442.

Figure 6.10 shows the \( MG_{ent} \) with the entropy of stability score. From the left
sub-figure, we can observe that for goodware families, the entropy of stability score seems equally distributed, while the dots of $MG_{ent}$ falls under the area of the inverse function, while for malware families, the entropy of stability score seems equally distributed, and both very stable and unstable features have low $MG_{ent}$. From the right sub-figure, we can observe that more features have larger entropy (more unstable) in malware families than goodware families. The p-value equals 0.038 in this analysis and we can make a strong claim of the stability on the entropy of stability score vs. $MG_{ent}$.

Figure 6.10: $MG_{ent}$ with the entropy of stability score

Figure 6.11 shows the $MG_{std}$ with the standard deviation of stability score. From the left sub-figure, we can observe that the standard deviation of all features’ stability
scores seems equally distributed in both goodware and malware families. Also, very few features have large $MG_{std}$, while most features have low $MG_{std}$. From the right sub-figure, we can observe that more features have a larger standard deviation (more unstable) in malware families than goodware families. Therefore, we can conclude that general features are more stable (with a larger standard deviation) in goodware families than in malware families according to the definition of $MG_{std}$, and the statistical analysis is significant with p-value equals to $3.182 \times 10^{-10}$.

(a) The standard deviation of stability score vs. $MG_{std}$  
(b) The standard deviation of stability score in Goodware vs. the standard deviation of stability score in Malware

Figure 6.11: $MG_{std}$ with the standard deviation of stability score

6.4.6. $MG_{\mu}$ ratio with different features

In this section, we conduct experiments on 3 $MG_{\mu}$ ratios on 4 different kinds of features (API Packages, Permission, Opcode, and System Command). We want to know if the $MG_{\mu}$ ratio can reveal the stability of all different types of features in the goodware and malware families. We have done many experiments on the $MG_{\mu}$ ratio in this section, but because of the page limitation, we only put some significant results in this section. In the following Section 6.4.7 we summarize all the results from these 3 different definitions of MG ratio and the stability analysis of different kinds of features.

Figure 6.12 shows the $MG_{avg}$ with mean stability score among API Package
features in goodware and malware families. From the left sub-figure, we can see that some features are relatively unstable in malware families with the stability score < 0.87, while all API features in goodware families have the stability score > 0.87. Also, we can claim that for API package features, features are more stable in goodware families than in malware families with p-value equals to 0.002 ≤ 0.05, which means the result is significant.

Figure 6.12: \( MG_{avg} \) with mean stability score among API Package features

Figure 6.13 shows the \( MG_{ent} \) with the entropy of the stability score among Permission features in goodware and malware families. From the left sub-figure, we observe that for goodware families, the entropy of stability score seems equally distributed, while the dots of the corresponding \( MG_{ent} \) fall under the area of the Inverse function. Again, we see that from the right sub-figure, most permission features in malware families have larger entropy than in the goodware families, which means that features are more stable in goodware families than in malware families in Permission features. The statistical analysis is conducted with a p-value equals to \( 3.352 \times 10^{-12} \), which means the result is significant.

Last, Figure 6.14 shows the \( MG_{std} \) with the standard deviation of the stability score among Opcode features in goodware and malware families. From the left sub-figure, we observe that the standard deviation of Opcode features’ stability score forming
SAAM: Stability Analysis of Android Malware Families

(a) Entropy of the stability score among Permission features vs. $MG_{ent}$

(b) Entropy of the stability score in Goodware vs. Entropy of the stability score in Malware among Permission features

Figure 6.13: $MG_{ent}$ with the entropy of the stability score among Permission features

two concentrated clusters, and most Opcode features in both goodware and malware families have $MG_{std}$ ranges from 1 to 1.75. Also, we see that from the right sub-figure, most Opcode features in malware families have a larger standard deviation than in the goodware families, which means that features are more stable in goodware families than in malware families in Opcode features. The significant analysis with $p$-value equals to $1.069 \times 10^{-26}$ also supports our claim.

(a) The standard deviation of stability score among Opcode features vs. $MG_{std}$

(b) The standard deviation of stability score in Goodware vs. the standard deviation of stability score in Malware among Permission features

Figure 6.14: $MG_{std}$ with the standard deviation of stability score among Opcode features
6.4.7. Conclusions

According to the extensive experiments we conduct on 3 different $MG_\mu$ ratio (stability score, entropy, and standard deviation) with 4 different features in Android goodware and malware families, we can make the following conclusions, where *Very strong* means all $MG_\mu$ support the conclusion, *Strong* means 2 of 3 $MG_\mu$ support the conclusion, *Weak* means only 1 $MG_\mu$ supports the conclusion.

**Conclusion 1.** There is strong evidence that general features are more stable in goodware families than in malware families.

**Conclusion 2.** There is weak evidence that API Package features are more stable in goodware families than in malware families.

**Conclusion 3.** There is very strong evidence that Permission features are more stable in goodware families than in malware families.

**Conclusion 4.** There is very strong evidence that Opcode features are more stable in goodware families than in malware families.

**Conclusion 5.** There is evidence that System Command features are equally stable in goodware families and malware families.
Chapter 7

Conclusions and Future Work

Section 7.1

Conclusions

With the rise in popularity of the Internet of Things, 5G, and other technology, mobile smart devices are increasingly evolving, and the number of Android apps mounted on smart terminals such as smartphones and tablets is growing. However, malware targeting the platform has risen as a result of this. As a consequence, there has been a lot of work done on detecting malware in Android apps. Artificial intelligence techniques like machine learning and deep learning have significantly increased the chances of detecting Android malware. This thesis introduces advanced and robust features for detecting specific Android malware families using data-driven analysis and characterization techniques. This thesis’s main goal is to provide a comprehensive summarization of the proposed effective Android malware detection systems and the Android malware feature stability analysis system.

First, we have presented DBank, a novel framework for distinguishing between Android banking trojans (ABTs) and goodware, and other types of malware. In particular, we propose a feature set based on the novel concept of Triadic Suspicion
Conclusions and Future Work

Graph (TSG). We show that, while we achieve similar accuracy to lightweight feature sets in past work, TSG-based features are more robust to some adversary attacks, and still achieve high accuracy even when using a subset of training data. We evaluate our system on recent (2016-2017) Android ABTs, and we show how DBank can automatically extract relevant features that can highlight differences from specific ABT families vs. goodware and other-malware. The concept of Triadic Suspicion Graph can not only be applied to the field of Android malware detection but can also be applied to other machine learning tasks when there is some hidden relationship between both the positive and negative samples. Our method may contribute to a more robust system with potential higher accuracy.

Second, we present a data-driven characterization of the principal factors that distinguish modern Android spyware (July 2016 - July 2017) both from goodware and other Android malware, using both traditional machine learning classifier and the proposed Ensemble Late Fusion (ELF) architecture that combines the results of multiple classifiers’ predicted probabilities to generate a final prediction. We show that ELF outperforms several of the best-known traditional and deep learning classifiers. The proposed Ensemble Late Fusion (ELF) architecture can always improve (at least lead to no compromise) the general machine learning classification results because of the mathematical attributes behind it, and at the same time, ELF can reveal the best classifier combination that should be used in different classification tasks according to the weight distribution of each classifier.

Third, our FARM (Android Malware Detection via (Somewhat) Robust Irreversible Feature Transformations) technique is novel and makes the following contributions: (i) we propose three new feature transformation techniques that can be used to generate feature vectors that are very hard to reverse engineer; (ii) we propose the FARM techniques that use these transforms to predict whether a given Android APK is a form of malware or not — we consider three forms of malware, namely spyware,
banking trojans and rooting malware. (iii) we propose three new kinds of attacks that
a malicious hacker might take to evade standard classifiers and show that FARM is
quite robust against these kinds of attacks. In particular, when there are no attacks,
FARM slightly outperforms various baselines and when these three attacks are used,
FARM is on average about 3 times more robust than the baselines. At the same
time, our work is not purely theoretical: FARM has discovered two Android APKs
to be rooting apps before any of the 61 anti-viruses on VirusTotal came to the same
conclusion. These samples were reported to Google’s Android Security Team who
have confirmed the labeling of these samples as rooting apps. The proposed FARM’s
feature transformations can not only be applied to the Android malware detection
system, but also on any set of “base” features to increase the robustness of the machine
learning-based system under certain types of attacks.

Last, we introduce SAAM (Stability Analysis of Android Malware Families) and
investigate how malware samples from the same family changing over time and the
efficiency of detection techniques at the feature level. In the meantime, we define
optimal-partition, stability score, and three different kinds of MG (malware over
goodware) ratio on the features over time in Android apk families. We also conduct
stability score analysis on 122 families of malware between 2012-2019 from VirusTotal,
and 120 families of goodware between 2012-2020 from Google Play Top free apps,
with an average of 60 samples for each family. Finally, we summarize 4 kinds of top
stable and unstable features over all the collected Android apk families, especially on
Android API features, Android Permission features, Operation Code features, and
System Command features. The definitions of optimal-partition, stability score, and
MG ratio can be applied to any field when the object under study can be transformed
into a time-continuous signal. After analyzing the signal, we may be able to link
to some external burst events from the key points in the signal, so that we can get
a better understanding of the signal itself. In addition, after analyzing the time-

115
Conclusions and Future Work

continuous signal, we are also able to summarize some effective information to aid machine learning classification tasks, thereby further improving the performance of the classifier.

Section 7.2

Limitations

This section presents limitations and suggestions for future improvement of the current work on Android malware detection and characterization.

First, the dynamic analysis we perform through the Koodous online service does not cover all possible execution traces. In particular, Koodous relies on the Droidbox sandbox, executes samples for 60 seconds, and collects any system and network activity detected during execution. Since the malware may detect that it is running in a sandbox and/or because its behavior may be triggered only by certain events (e.g., initiated by the attacker’s command and control server), some dynamic behavior may be missed. Very recent work studies in-depth how to trigger malicious behaviors of malware by simulating user interactions, but the problem remains an open issue and future work could look at integrating these approaches into dynamic analysis.

Second, it is worth observing that some downloaded Android apk samples crashed during dynamic analysis on Koodous and therefore are not included in the analysis in this thesis. While this could indicate that our study is not complete, this limitation is common in prior work that uses dynamic analysis to study Android malware. Some strategies to prevent crashing of samples may be worth investigating in future work.

Third, the feature transformation techniques also have some limitations that we discuss briefly below. Though FARM’s feature transformations can be applied to any set of “base” features, it is important that this set of base features be selected judiciously and be capable of making good predictions. In this thesis, we chose base
features that have been shown in the literature to be useful for classifying Android apps into benign vs. malicious samples. Also, we have shown that FARM is robust against three types of attack. However, there may be other kinds of attacks (e.g. attacks that do not depend on API function calls or permissions that we have not tested against). While we do not expect to find classifiers that are robust against every type of attack, identifying a larger space of attacks and showing how FARM either is robust to those attacks or could be modified to withstand those attacks is an important future research topic.

Section 7.3

Future Work

In this thesis, we characterize Android Banking Trojans, Spyware, and rooting malware with respect to goodware and to other malware. We rely on machine learning features derived from lightweight static and dynamic analysis of Android samples. In future work, it would be interesting to consider additional techniques to gain more insights into specific Android malware families. Examples of such techniques include (i) designing new features to represent sequence and timing of dynamic operations; (ii) integrating dynamic analysis techniques for detecting which specific information is likely leaked through different channels (e.g. network, SMS, files on disk) through which such leaks occur; (iii) performing in-depth inspections of application source code and control-flow to link the malware behavior to the code; (iv) considering more real cases of the adversarial model in which the adversaries tampers with our training data (poisoning) or performs test-time evasion attacks; (v) Another interesting research direction is to explore multi-label classification—for instance, acecard is both spyware and a banking trojan, and a system that can automatically predict a set of labels for each app would be useful for capturing multiple malicious behaviors at once.
Bibliography


[7] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre


[28] Luyi Xing, Xiaorui Pan, Rui Wang, Kan Yuan, and XiaoFeng Wang. Upgrading


[36] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad


[90] Andrea Possemato, Andrea Lanzi, Simon Pak Ho Chung, Wenke Lee, and Yanick Fratantonio. Clickshield: Are you hiding something? towards eradicating click-


